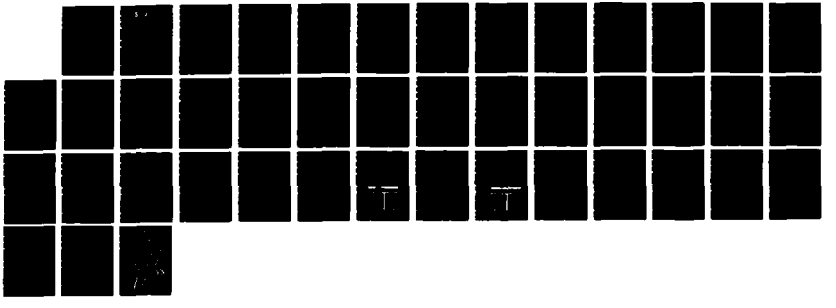
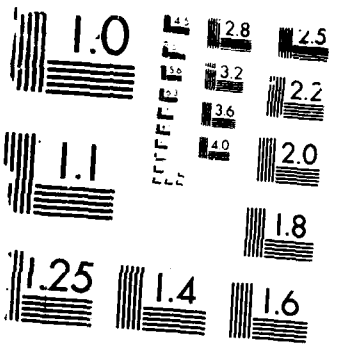
AD-A188 499 A LABORATORY FACILITY FOR RESEARCH IN PARALLEL COMPUTATION: PROJECT FINAL (U) INDIANA UNIV AT BLOOMINGTON DEPT OF COMPUTER SCIENCE D GANNON JUL 87 1/1

UNCLASSIFIED AFOSR-TR-87-1981 AFOSR-86-0279 F/G 12/6 NL





RESOLUTION TEST CHART

SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED

AD-A188 499

DTIC FILE COPY

ELECTE JAN 20 1988 S D

1a REPORT SECURITY CLASSIFICATION	
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 87-1981
6a NAME OF PERFORMING ORGANIZATION: Indiana University Foundation	6b OFFICE SYMBOL (If applicable):
6c ADDRESS (City, State, and ZIP Code): Bloomington, IN	7a NAME OF MONITORING ORGANIZATION: AFOSR/NM
	7b ADDRESS (City, State, and ZIP Code): AFOSR/NM, Bldg 410, Bolling AFB DC 20332-6448
8a NAME OF FUNDING/SPONSORING ORGANIZATION: AFOSR	8b OFFICE SYMBOL (If applicable): NM
	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: AFOSR-86-0279
8c ADDRESS (City, State, and ZIP Code): AFOSR/NM, Bldg 410, Bolling AFB DC 20332-6448	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. 61102F; PROJECT NO. 2304; TASK NO A3; WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)

A Labortory Facility for Research in Parallel Computation: Project Final Report

12 PERSONAL AUTHOR(S)
Dennis Gannon

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Final	FROM 7/31/86 TO 7/30/87	Jul 87	40

16 SUPPLEMENTARY NOTATION

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This DOD URIP effort provided resources for the purchase of a BBN Butterfly parallel processor architecture for research into parallel processing. The equipment has been used to developed parallel algorithms for ray traced computer graphics, for numerical fast Fourier Transform (FFT) algorithms, and AI and expert system applications. Papers produced include such titles as "Distributed Genetric Algorithms", and "A software tool for Building Supercomputer Applications".

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	
22a NAME OF RESPONSIBLE INDIVIDUAL: Maj. John Thomas	22b TELEPHONE (Include Area Code): (202) 767-5026 / 22c OFFICE SYMBOL: NM

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED

A Laboratory Facilitiy for Research in Parallel Computation: Project Final Report.

Dennis Gannon

Dept. of Computer Science Indiana, University Bloomington, Indiana

ABSTRACT

This report describes the activities carried out under AFOSR GRANT 86-0279 from the starting date to ~~Oct. 1, 1987~~ 31 Jul 86.

1. INTRODUCTION

It has become a certainty that Multiple Instruction Stream, Multiple Data Stream (MIMD) parallel architectures are going to play a major role in all aspects of high speed computer design for the foreseeable future. The technology needed to build these machines is very well understood and many commercial systems are now available. However a major problem still must be solved before we can say that parallel computation is the wave of the future. The problem is due to the fact that each machine represents a different model of parallel computation. These difference are reflected not only in the software but in the very way one designs algorithms to best use the hardware. Consequently, it is a major challenge to get programs to run on these machines and when one is working on one machine it can be a major effort to get it to work on a different machine. In our research projects we have focused on this portability issue from the perspective of parallel algorithm design and how it effects the internal organization of advanced compilers.

Parallel computation has both a theoretical foundation and an experimental component. It is a science where experience and experiment drive the formulation of new theories. Because we are interested in both the process of porting programs and the behavior of the algorithms on different target machines, it was essential to build a laboratory for parallel computation. With help from this grant from the University Instrumentation Program (and nearly $100,000 from Indiana University) we built a parallel computation research lab. The lab houses two machines of great interest to us. One is a 16 Processor BBN Butterfly parallel shared memory computer. The other is an Alliant FX/8 4 CE vector multiprocessor. These two systems provides an outstanding basis for research on the portability problem because they have radically different architectures but are both still classified as shared memory parallel computers.

In this report we will first describe some of our algorithm research and then we will focus on the compiler design problems.

2. ALGORITHM EXPERIMENTS

To understand to issues involved in porting parallel algorithms from one parallel machine to another it is essential spend a considerable amount of energy doing just that. Our target machines include the two systems that we operate in our lab. One is a 16 processor butterfly shared memory computer and the other is an Alliant FX/8 with 4 vector processors. There are four experiments described here. Each involves the design of a parallel algorithm and in most cases a complete application package. In each case a version for both machines has been completed or a version for one machine is complete and the version for the other is still in development.

Ray Traced Computer Graphics.

This experiment was carried out to test the problems of extracting the parallelism in an application that is very computationally intensive but also has data structures that are more closely associated with recursive algorithms than the traditional numerical codes. The algorithm works by following optics in reverse. Light rays are "traced" from the eye of the viewer back into the scene where they reflect off and refract through objects. Because each ray is independent of all of the others the task is completely parallel. One processor can be assigned to each light ray and massive parallelism can be obtained. This was done by a team of students for the butterfly and reasonably good performance resulted.

The primary problem was that the object data base was stored in globally shared memory and each processor needed constant access to this data base. Because a global memory module can only be reference at constant rate and only one processor may have access to the data in that memory module at any given instant of time, there is an upper bound on the number of processors that may share an object that they frequently reference without causing some conflicts and delays. By distributing the shared data through the set of memory memory modules in a uniform manner, we were able to reduce the contention and increase performance.

An important lesson was learned here. For large shared memory parallel systems the distribution of data can, and must, be a major task of any compiler that tries to optimize performance.

The Alliant FX/8 presented a different set of problems. First, the processor on the Alliant machine contains complex and powerful vector hardware. The problem is that it is not easy to exploit on this algorithm. We did, however, discover that there are a number of ways that it can be exploited for simple computations that must be carried out for each ray. For example, each ray must be intersected with each object in the scene (for a simple ray tracing algorithm). This process may be easily vectorized and good performance results. For more complex algorithms this task is not needed. We are still studying the problem of how to provide effective exploitation of vector hardware on the problem.

Numerical FFT algorithms.

Numerical FFTs are just one of many numerical computations that we have worked on. In all cases we have found one striking difference between the effective use of the parallel hardware one our two systems. In particular, we have discovered that on the Alliant system the memory hierarchy is such that processors "like" to share common data (because it may be kept in the shared cache). Furthermore, because the cost of bringing data into cache from shared memory is relatively high, it is best to try to make sure that all required references to a data item by all processors occurs while the data is in cache. While this may seem obvious, it has strong implications about the way algorithms are organized. In fact, Jalby, Meyer, and Gallivan have shown that a block structured algorithms achieve the best performance on the machine. Based on their results we (Jalby and Gannon) designed an FFT library for the Alliant that is very fast and we are now incorporating the block structuring transformations into the programming tools system.

On the Butterfly there is no shared cache and no strong need to do blocking. However, there is a related problem and solution. The memory on the butterfly is local. This means that when data is in a local memory the access is much faster than if it is far away. We discovered that the same analysis that was needed to keep data in cache for the Alliant could be used to decide which data must be kept in the local memory of each Butterfly processor. In fact, we have just been informed that after reading our paper, the people at BBN have decided to try to implement our FFT algorithms as part of a package of numerical tools for the Butterfly.

A.I. Production Systems. OPS 5.

Two other algorithm application areas that being studied in our laboratory are related to Artificial Intelligence and Expert Systems. One is Neural Network Modeling which will not be described here and the other is a family experiments with the production system language OPS5. Production systems are used in the inference engines of expert systems. One of the most common is OPS5 and it is based on a tree resolution method called the Rete Match algorithm. We have now completed one implementation based on using butterfly Lisp on the BBN system. This proved to be far too slow partially due to compiler problems, but mostly due to the fact that the obvious ways to try to use concurrency in the match algorithm do not work. (This fact has been reported by several people in the literature).

We have started a new effort that will focus more energy on the lower levels of the computation that should prove to be effective for both the Alliant system and the Butterfly. The new version of the system is based on a redesign of the OPS5 language to allow the full evaluation of Scheme expressions in OPS5 productions and the whole system has been recoded in C to provide a fast implementation on both the Butterfly and the Alliant. This project is the work of graduate students Lawrence Tenny and Charles Daffinger and will require at least another year of experimentation before we can form any conclusions.

Genetic Algorithms.

This work is being done by graduate student J. Y. Suh under the direction of faculty member Dirk Van Gucht. Genetic algorithms are an optimization technique that uses simple ideas from evolution theory to solve optimization problems. In this exercise we started with a good serial C program for doing a genetic optimization of the traveling salesman problem. We then did a mechanical set of transformations to come up with a reasonably good Butterfly version. A series of test were made which showed moderate performance improvements that resulted in speed-us of about 12 on a 16 processor machine.

By looking at where the restructured algorithm failed to perform with perfect speed-up he noticed that the serial algorithm was bound by a centralized control mechanism that inhibited parallelism. By focusing on this problem was able to design an completely new "distributed" genetic code. The new code has been run on Butterfly systems with as many as 128 processors with speed-ups of over 120. A copy of a summary technical report on this work is attached to this report.

3 . ADVANCED COMPILER DESIGN

We feel that our attempts to help automate the process of restructuring serial program are going very well and that these tools that we are building are essential if we have any hope of solving the portablility problem. However, one clear message that has emerged from our experimental work. It is not possible to derive the OPTIMAL algorithms for any given computation by a purely mechanical set of transformations to the source code of a good serial algorithm. Algorithm RETHINKING is needed to do that. The important question to ask is what sort of tools are needed to help programmers with this process.

Again, it is our experimental work that has led us to an answer to this question. The basic procedure that programmers usually follow when trying to design a new parallel algorithm is that they to try to discover exactly why the old one failed. The programmer does this by testing the program and isolating the serial bottlenecks in the computation and understanding why they take the form that they do. It is in this simple process where programmer need the most help.

In particular, we have found that programmers have the greatest difficulty when the code seems to have ample parallelism, but for some unknown reason, the machine fails to deliver the speed-up that was expected. This problem is more common than the problems associated with lack of concurrency in the code. The concurrency is there, but for some reason the machine can not use it. Often the solution has to do with the fact the the "granularity" of the computation is ill suited to the hardware or, more often, the loss of performance is due to problems in the way the algorithm exploits the memory hierarchy.

So far we have focused on the properties of memory hierarchy such as cache memory and processor local memory. We have developed a mathematical model of how cache behavior can be related to program data dependencies. This work was done in collaboration with William Jalby of INRIA in Paris and Kyle Gallivan of CSRD in Urbana. The results were presented in an invited paper in the first international conference on supercomputing in Athens Greece in July of 1987. The next step is to design a mathematical model of task granularity and synchronization. We are still working on this problem.

Part of our work on this problem has led to the first automatic restructuring compiler for the BBN butterfly. Graduate student Mannho Lee has built a portable runtime environment that can be used on both the Butterfly and the Alliant and a code generator that translates the data dependence graph coming out of our program restructure into a C program that can execute in parallel with the aid of the runtime environment. A copy of a technical report describing this work is attached to this document. This report will be presented at the SIAM Parallel Processing Conference in Dec. 1987.

Distributed Genetic Algorithms

Jung Y. Suh
Dirk Van Gucht

Computer Science Department
Indiana University
Bloomington, Indiana 47405
(812) 335-6429
CSNET: jvsuh@indiana, vgucht@indiana

1. Introduction

1.1. Genetic Algorithms

Suppose we have an *object space* X and a function $f : X \rightarrow R^+$ (R^+ denotes the positive real numbers) and our task is to find a global optimum for that function. *Genetic algorithms* are a class of adaptive algorithms invented by John Holland [16] to solve (or partially solve) such problems. Genetic algorithms differ from more standard search algorithms (e.g., gradient descent, controlled random search, hill-climbing, simulated annealing [3, 4, 18] etc.) in that the search is conducted using the information of a *population* of *structures* of the object space X instead of that of a single structure. The motivation for this approach is that by considering many structures as potential candidate solutions, the risk of getting trapped in a local optimum is greatly reduced. In Figure 1 we show the layout of a genetic algorithm, which we will from now on call a *standard genetic algorithm*.

```
P(t) denotes the population at time t.

t ← 0;
initialize P(t);
evaluate P(t);
while (termination condition is not satisfied)
{
   t ← t+1;
   select P(t);
   recombine P(t);
   evaluate P(t);
}
```

Figure 1. Layout of a Standard Genetic Algorithm

The initial population P(0) consists of structures of X, usually chosen at random. Alternatively, P(0) may contain heuristically chosen structures. In either case, the initial population should contain a wide variety of structures. Each structure x in P(0) is then evaluated by applying to it the function f. The genetic algorithm then enters a loop. Each iteration of that loop is called a *generation*. The new population P(t+1) is constructed in two steps, the selection and recombination steps. In the selection step, a temporary population (say $P'(t+1)$) is constructed by choosing structures in P(t) according to their relative performance. For example, if we are maximizing f, the structures with greater than average performance will be selected with higher probability than the structures with below average performance. This resembles the *survival of the*

fittest principle of natural evolution. After the selection step, the temporary population $P'(t+1)$ is *recombined*. (The resulting population is the new population $P(t+1)$.) Typically, recombination is accomplished by applying several *recombination operators*, such as *crossover*, *mutation*, *inversion* [7, 16], or *local improvement* [25], to the structures in $P'(t+1)$. After the recombination step is completed, the new population is reevaluated and a termination condition is checked for validity.

Genetic algorithms have been applied with great success by De Jong [7] to a wide variety of function optimization problems defined over object spaces of the form R^n, i.e., each structure x consists of n real numbers $x[1] \ldots x[n]$. They have also been applied to other problems such as optimization of simulations [12], image processing tasks [10], evolving production system programs for AI [24], combinatorial optimization problems [5, 6, 11, 14, 15, 23, 25] etc. This wide variety of problem domains suggests that genetic algorithms are robust and flexible optimization algorithms. They suffer a serious drawback however: their implementation as sequential algorithms on sequential machines typically run slowly when compared to problem specific optimization algorithms. On the other hand, it is clear, by looking at Figure 1, that genetic algorithms can easily be parallelized and run on multi-processor machines which would greatly improve their efficiency. It is our intent to show a variety of parallel versions of genetic algorithms, which, by the way, better resemble natural evolution, and to show the results and measured speed-up of running these algorithms as simulations on sequential machines and as real parallel algorithms on a multi-processor machine.

1.2. Parallelizing Genetic Algorithms

There have been many proposals to improve the quality and performance of standard genetic algorithms. Many of these are intended to improve the robustness of the algorithm, mainly by preventing the *premature convergence problem* [7, 8, 16] by maintaining enough diversity in the population, either by normalizing the performance value of the structures in the population [1] or by introducing random noise systematically [21]. Another improvement of genetic algorithms resulted from the proposals indicating that domain specific knowledge could easily be incorporated in the recombination operators of genetic algorithms [14, 15, 25]. Still another proposal indicating that it is sometimes sufficient to provide approximate, but typically quickly obtained, function evaluations to the algorithm resulted in dramatic speed-ups of the algorithm in problem domains such as image processing [10]. In this paper, we readdress the problem of speeding up genetic algorithms by parallelizing them†. As one can observe, by looking at Figure 1, one can easily devise a parallel version of a standard genetic algorithm by considering a pool of processors which perform function evaluations and recombination operations and another processor which is responsible for assigning structures to the processors for evaluation and recombination and which furthermore performs the selection step of the genetic algorithm (in fact this is close to one of the algorithms introduced by Grefenstette [13]). We will show that one can go a lot further by also parallelizing the recombination step and the selection step of the algorithm. In Section 2, we propose a parallel version of a standard genetic algorithm in which the evaluation step and the recombination step are parallelized. We will call this algorithm the *centralized genetic algorithm* since it still uses central control because the selection step is performed by a master processor which also synchronizes the actions of the processors which perform the evaluations and recombination operations. In Section 3, we propose a framework in which genetic algorithms become totally distributed algorithms which we will call *distributed genetic algorithms*. This is accomplished by replacing the selection step by local selection routines which are distributed over the processors which already contain routines for evaluation and recombination. We will furthermore argue in Section 3 that distributed genetic algorithms yield similar performance as standard genetic algorithms, that their implementation is straightforward, uniform and natural, that they are reliable algorithms, that they allow for effects

† This problem has been considered by other researchers such as Grefenstette [13].

not possible in synchronized implementations, and that they offer more tuning opportunities to control problems, such as the premature convergence problem, than standard genetic algorithms. In Section 4 we compare the centralized and distributed genetic algorithms with parallel versions of genetic algorithms introduced by Grefenstette [13]. In Section 5, we provide experimental results of centralized and distributed genetic algorithms for the *traveling salesman problem* and show that their performance is as good as standard genetic algorithms but that they run faster due to the speed-up obtained from the parallelism. Finally, in Section 6, we draw some conclusions and discuss some directions of future research.

2. Centralized Genetic Algorithms

In this section, we present an algorithm which parallelizes the standard genetic algorithm shown in Figure 1 and comment on some of its shortcomings.

Consider a pool of (identical) processors (called *slave processors*) which each contain a structure of the population and which can evaluate structures and perform recombination operations, such as cross-over, mutation or local improvement, and consider another processor (called the *master processor*) whose task it is to instigate and synchronize the evaluation and recombination steps and to perform the selection step. In Figure 2 we show the code executed by a slave processor and the code executed by the master processor.

```
                    SLAVE PROCESSOR

{
    if the master processor requests evaluation then
        evaluate the local structure;
    if the master processor requests recombination then
        perform recombination to the assigned structures;
}

                    MASTER PROCESSOR

{
    while termination condition does not hold
        {
            while any slave is active WAIT;
            perform (global) selection (this involves
              reassigning structures to slave processors);
            request evaluation from the slave processors;
            request recombination from the slave processors on
              assigned structures;
        }
}
```

Figure 2. A Centralized Genetic Algorithm.

As can be seen, parallelization and the accompanying speed-up is achieved by distributing the work required in the evaluation and recombination step over the slave processors. Notice, however, that the selection step is central to the task of the master processor. In fact, stated from a different perspective, it is because of the selection step that a master processor is necessary. This

is the case because selection, as described in the current literature, is a global process requiring knowledge about the values of all structures of the current population. It is for this reason that we call this algorithm a *centralized genetic algorithm.* Although, this algorithm is a natural parallel implementation of a standard genetic algorithm and achieves the speed-up it is designed for, it has disadvantages:

i. *the algorithm is not reliable*: indeed, as can be seen from the code shown in Figure 2, if the master processor or one of the slave processors fails, the algorithm halts.

ii. *synchronization delays may occur* because evaluation and recombination operations may not all require the same time when applied to different structures and therefore processor time is wasted in the form of idle time.

iii. *the algorithm does not appear natural because selection is centralized.* It seems to us that, in a broader sense, selection is a process that should not be centralized to a single processor; it certainly is not implemented as such in nature. In fact, in nature, selection, as a global effect, is achieved through the continuous interaction and competition of individual structures and is not controlled by a central agent. In Section 3 we will see how to overcome this very problem and obtain a more natural and uniform parallel genetic algorithm. It should also be noted that a slow-down of the algorithms can be expected because selection is not parallelized as opposed to evaluation and recombination.

3. Distributed Genetic Algorithm

In this section, we propose a framework in which the principal components of genetic algorithms can be implemented as local processes. We then argue why we think this implementation is more natural and at least as efficient as the standard implementation. Our framework consists of a pool of processors which execute identical or nearly identical tasks in parallel. Each processor has a local memory large enough to store a small number of structures, one of which will be called the *local structure*. The collection of all these local structures in the processors constitutes the population of the genetic algorithm, hence rather than a having a global memory to store the structures of the population, the structure are spread out over the processors in the form of local structures of processors. Furthermore, each processor is capable of performing local tasks and communicating with the other processors. In this framework, we can describe a new way of implementing genetic algorithms. As indicated before (see Figure 1), a genetic algorithm breaks down into the repeated application of an evaluation step, a recombination step and a selection step. It is straightforward to implement the evaluation and recombination steps. In the evaluation step, each processor evaluates its local structure and stores the outcome in its local memory. The recombination step which usually consists of a cross-over step and a local improvement step is implemented as follows:

i. For the cross-over step, each processor p elects to communicate with another processor q, with some locally controlled probability. After communication is established, processor p reads in the local structure of q, after which communication between p and q ceases to exists. Processor now performs cross-over between the structure just read in and its local structure and one of the offsprings becomes the new local structure of p.

ii. For the local improvement step, each processor probabilistically determines to perform local improvement on its local structure.

The novelty of our approach comes from the fact that we also propose to implement the *selection* step by local processes. In the standard genetic algorithm, the selection step is implemented by a single process which gathers the performance value of the structures, computes their average and "duplicates" the structures according to their relative performance with that average. If one wants to faithfully replicate this process, one has to introduce a special processor for this step of the genetic algorithm. In our opinion, this is unnatural as well as unnecessary. It is unnatural

since we do not believe in a supervising agent which, for each structure, assigns its rating and calculates the number of offsprings (certainly, nature does not seem to behave that way). It is also unnecessary since selection can be implemented, as will be seen shortly, by local processes. There are several ways to implement a selection step using local processes. What is common to all of them, though, is that they all implement a notion of the survival of the fittest principle. We next outline five different, but related, selection steps, called *Selection 1, Selection 2, Selection 3, Selection 4 and Selection 5.*

In Selection 1, each processor p, with some locally controlled probability elects to communicate with another processor q, if the value of the local structure of p is better than the value of the local structure of q, processor p overwrites the local structure of q with its local structure after which communication is ceased, otherwise p undertakes no action and communication is ceased immediately (notice that processor q is passive in this process). In Selection 2, each processor p, with some locally controlled probability elects to communicate (not necessarily simultaneously) with k other processors $q_i (1 \leq i \leq k)$, p reads in the value v_i of the local structure of q_i and stores the processor number of q_i and ceases communication with q_i. Processor p computes $av = 1/k \sum_{i=1}^{k} v_i$, the average value of the v_i's, and compares the value v of its local structure with av. If $v > av$ then p randomly selects another processor q and overwrites the local structure of q with its local structure, otherwise p undertakes no further action. In Selection 3, the following action is undertaken by processor p: if $v > av$ then p randomly selects one of the processors q_i (remember p has the processor numbers of the q_i's in its local memory) and overwrites the local structure of q_i, otherwise p undertakes no further action. In Selection 4, the following action is undertaken by processor p: if $v > av$ then p overwrites the local structure of the processor among the k processors q_i with the worst v_i-value, otherwise p undertakes no further action. In Selection 5, the following action is undertaken by processor p, if $v > av$ then p overwrites the local structures of l processors, with $l = \lceil min(k, v/av) \rceil$, of the selected processors q_i, otherwise p undertakes no further action. It is interesting to notice that Selection 1 is a special case of Selection 4 and Selection 5 for $k = 1$ and that Selection 5 is quite related to the standard selection step of sequential genetic algorithms. It should also be noted that we can easily incorporate normalization techniques as suggested by Baker [1] within these selection schemes.

In Figure 3 we summarize the above discussion by showing the code each processor executes during the course of a run of a distributed genetic algorithm. Notice that we do not require that a processor executes the four statements in the while loop in the specified order or that p_l, p_c or p_s are the same for all processors.

```
{
    while termination condition does not hold
        {
            evaluate the local structure;
            perform local improvement on the
              local structure with probability p_l;
            perform cross-over with probability p_c;
            perform local selection with probability p_s;
        }
}
```

Figure 3. A Typical Processor of a Distributed Genetic Algorithm.

We are now ready to give a description of a distributed genetic algorithm. A *distributed genetic algorithm* (DGA) consists of a pool of processors as described above which are initialized by assigning to each of them a local structure and are then run asynchronously with each processor

executing its local code as shown in Figure 3. The DGA adopts the following synchronization policy: if a processor p wants to communicate with another processor q, p places a lock on q which is released when communication between the two processors ceases; if during this communication another processor r wants to communicate with q, communication between r and q is not granted and r proceeds by trying to communicate with another processor. Notice that this simple synchronization policy can be implemented by local processes as well.

There are certain observations we want to make about distributed genetic algorithms:

i. *they yield similar performance as standard genetic algorithms*: experiments with DGAs on function optimization problems and combinatorial optimization problems yielded performance, both in speed and in robustness, of the same quality as experimental results with the standard sequential genetic algorithms reported in the literature. In Section 4, we compare the performance of a DGA and a standard genetic algorithm for the traveling salesman problem.

ii. *their implementation is straightforward and uniform due to the introduction of the local selection process*: in standard genetic algorithms, the selection step is a globally controlled process. This results in an asymmetry in their implementation since selection has to be considered separately from the evaluation step and the recombination step. In the distributed version, this asymmetry is removed and uniformity is obtained by localizing the selection step and therefore localizing all major components of the genetic algorithm. The global effect of a genetic algorithm is obtained because the processors communicate when performing crossover and selection. From an implementation point of view, also notice that we do not need sophisticated locking and scheduling mechanisms and that there is only a minimal contention problem [22] since processors rarely will be competing for the same memory locations.

iii. *their implementation is more natural*: in our opinion, the distributed genetic algorithm resembles closer the evolutionary process found in nature. In nature a pool of structures communicate and operate on each other in the form of local processes to yield the effect known as the evolutionary process. It does not appear likely that there is a supervising agent which controls this process or even parts of this process such as the selection step. In fact, we strongly believe that selection in nature is achieved through local processes which perform a kind of survival of the fittest strategy.

iv. *they are very reliable algorithms*: the failure of a processor only slightly alters the flow of the algorithm, in the worst case, the processor that fails has a lock on another processor and therefore disables that processor upon failure, but this will not have a major effect on the communication and the actions of the other processors, resulting in only a minor change in the flow of the entire algorithm. It should also be noticed that is is very easy to repair or insert processors without affecting the algorithm much.

v. *they allow for effects not possible in synchronized implementations of genetic algorithms*: due to the asynchronous behavior of the algorithm, different processors may display different behaviors. For example assume we have a processor g with a "good" local structure and a processor b with a "bad" local structure. It is likely that processor b will spend more time improving its structure by performing local improvements on its local structure, whereas processor g may in the mean time spend his time communicating with other processors through crossover or selection. Clearly this effect is by-passed in synchronized implementations of genetic algorithms such as the centralized genetic algorithms.

vi. *they offer more tuning opportunities*: since the crossover probability p_c, the local improvement probability p_l, the selection probability p_s, as well as the actual crossover, local improvement and selection routines are local to each individual processor (in contrast, in the standard genetic algorithms all these parameters and routines are the same) a DGA allows for more tuning opportunities by setting these parameters and operators not necessarily equal in all the

processors. It is, for example, quite likely that the parameters should change over the course of the algorithm and may change according to the properties of the local structure of each processor. This ability offers, for example, additional techniques to overcome the premature convergence problem found in most genetic algorithms. In fact, we have already observed this phenomenon in our experiments with DGAs.

4. Comparison with Other Parallel Implementations of Genetic Algorithms

There have been other proposals to parallelize genetic algorithms, the most noticeable among these, the proposal of Grefenstette [13]. We will state Grefenstette's assumptions, give two of his parallel genetic algorithms and along the way, compare and contrast his approach with ours.

Grefenstette's main assumption is that the dominant cost in a genetic algorithm is the amount of time spent in doing function evaluations. In other words, he assumes that the evaluation step takes the most time and the recombination and selection steps are merely small overhead. While this is a reasonable assumption in some applications, such as optimizations of simulations [12] and evolving production system programs for AI tasks [24], this assumptions is not valid in other applications such as some combinatorial optimizations problems like the traveling salesman problem or puzzle problems such as the sliding puzzle problem [25], where in fact as much time or even more time is spent in the recombination step as in the evaluation step due to the incorporation of heuristics in the crossover and local improvement operators [14, 15, 25]. Given Grefenstette's assumption, it is difficult to compare his algorithms with the distributed genetic algorithm, but it is still interesting to contrast both approaches. We state two of his algorithms next. *Algorithm 1* is an algorithm with a centralized concurrency control mechanism (we show this algorithm in Figure 4). Much like the CGA described in Section 3 it consist of $k + 1$ processors, one master processor and k slave processors. The master process maintains the population of structures and performs the selection and recombination steps of the genetic algorithm. The slave processors are responsible for structure evaluation.

Comparing Algorithm 1 and the CGA is left up to the reader. Algorithm 1 and the DGA basically coincide in that they both distribute function evaluations but greatly differ in the way recombination and selection is performed. In Algorithm 1, the master processor is responsible for these processes, in the DGA, recombination and selection are distributed in the form of local processes. As mentioned by Grefenstette, Algorithm 1 has rather poor reliability characteristics. If the master process fails, the entire algorithm halts. Furthermore, the synchronization mechanism employed relies on the fact that all slave processors successfully complete their actions. As mentioned in Section 3, the DGA is highly reliable, i.e., failure of a processor does only marginally affect the performance of the entire algorithm.

Algorithm 2 uses distributed, asynchronous concurrency control. There are k identical processors, one of them is shown Figure 5.

Although this algorithm is closer to a DGA since the evaluation and the recombination steps are distributed, it differs from a DGA in two ways:

i. selection is not localized since each processor has to update the selection probabilities of all structures of the population, and

ii. Algorithm 2 does not distribute memory, instead there is one global memory which stores the structures of the population. This can lead to contention problems as indicated by Grefenstette and can thus result in a slow-down of the algorithm. In contrast, in the DGA, there is no notion of a global memory which stores the population of structures, rather, the pool of processors with their local structures serves in the role of the population of the genetic algorithm. As indicated in Section 3, this implies that a simple locking mechanism with little contention problem suffices to implement successful communication, resulting in maximal speed-up given

```
                    SLAVE PROCESSOR

{
    while there are unevaluated structures in the population
        {
            choose a subset s_j1,...,s_jn
             of size n (where n = (size of the population)/k)
             from the set of unevaluated structures
             in the population;
            evaluate each of the chosen structures;
        }
}

                    MASTER PROCESSOR

{
    while termination condition does not hold
        {
            while any slave is active WAIT;
            perform the (global) selection step;
            perform the recombination step;
        }
}
```

Figure 4. Algorithm 1 of Grefenstette.

```
{
    while termination condition does not hold
        {
            remove n unevaluated structures from the population;
            evaluate the chosen structures;
            recombine the chosen structures;
            { enter critical section
                insert the structures into the population;
                update the selection probabilities;
            } leave critical section
        }
}
```

Figure 5. Algorithm 2 of Grefenstette.

an implementation on a multi-processor machine.

5. Experimental Results with Centralized and Distributed Genetic Algorithms

In this section, we present two sets of experimental results about centralized and distributed genetic algorithms. In the first set, we compare the results of a simulation of a CGA and a DGA on a sequential machine, a VAX 8800. In the second set, we compare the results of implementations of both algorithms on a multi-processor machine, a Butterfly machine with 16 processors [2].

The algorithms are applied to the Traveling Salesman Problem (TSP) [19, 20]. Those who are not familiar with this application, we refer to [11, 14, 15, 25] where standard genetic algorithms

are described to (approximately) solve this problem. Before discussing the results in detail, we would like to point out that genetic algorithms for the TSP do not satisfy the central assumption of Grefenstette [13]. As mentioned in Section 4, Grefenstette assumed that the evaluation of structure takes more time than the recombination operators. But, in case of genetic algorithms for the TSP, the recombination operators, i.e., the cross-over and local improvement operators, take as much as or more time than the evaluation.

5.1 Simulation Results on a Sequential Machine

Three different TSP problems were analyzed, their names and definitions are shown in Figure 6.

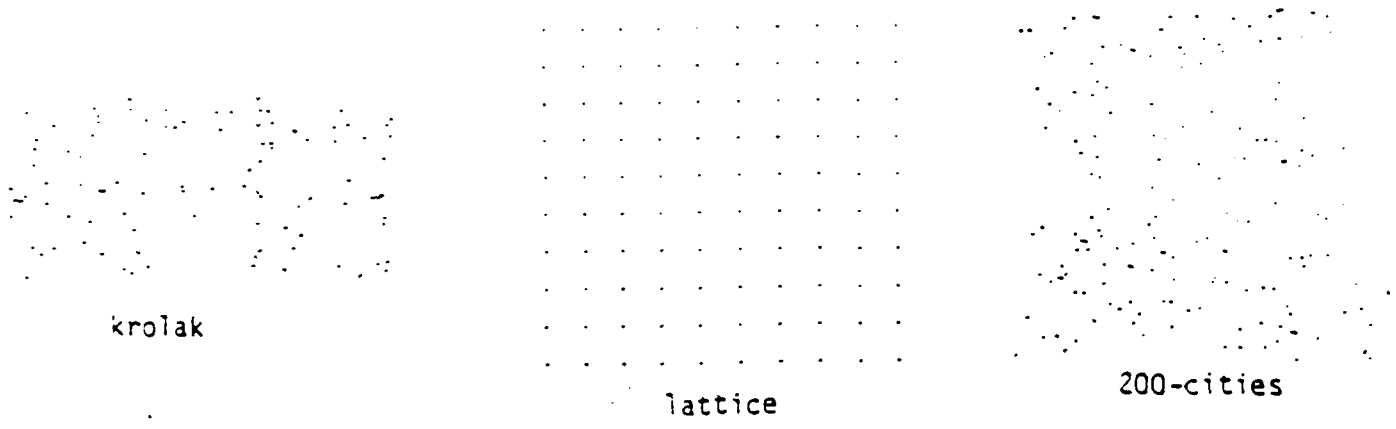


Figure 6. Three Traveling Salesman Problems.

The simulation of the CGA corresponds exactly to the standard GA. In the simulation of the DGA, we used Selection 1 as the local selection procedure, i.e., if processor p (with a certain probability p_s) elects to communicate with a processor q then if the value of the local structure of p is better than the value of the local structure of q, processor p overwrites the local structure of q with its local structure.

In Appendix 1 we show the results of running the simulations of the above described DGA and CGA. The most important observation is that the performance of the DGA which uses a local selection method is similar to that of the CGA which uses the standard selection method. This result indicates that it is possible to safely use the more natural distributed genetic algorithms and still obtain similar results. If there was a difference in the performance of the two algorithms, it was in the fact that local selection seems to add another source of maintaining the diversity because of its more noisy behavior. As indicated in previous work, this can only add to the robustness of the algorithm. In fact the noisyness of the local selection allowed us to use fewer local improvements (mutations) than was necessary for the CGA.

5.2 Some Parallel Implementation Results

The experiments described in this section were done on a Butterfly [2] machine with 14 available processors. Only one of three traveling salesman problems, the Lattice problem with 20, 60, and 100 cities was looked at in these experiments (we expect similar results for other TSP problems).

We ran each algorithm with a varying number of processors. This requires some comments because our algorithms are originally designed in such a way that one processor holds only one structure. What we mean by this is that, if we run for example a DGA with 100 structures on 2 processors, we allocate 50 structures on each processor and one processor sequentially simulate the task of 50 processors. So even in case of 14 processors running, each processor has to simulate 7 or 8 processes sequentially. The ideal case would be to run 100 processors in parallel. The purpose of running the algorithm on a varying number of processors is to find out how far the algorithm is parallelizable by deriving the curve of "effective processors". In most cases, using k processors does not result in k-times speed-up due to the overhead of communication or a section of program which cannot be parallelized. If we run a program with a single processor and then run it with k processors, we can find out how much speed-up k processors produced, by dividing the execution time of single processor run by that of k processors. For each k, we can derive the speed-up factor and draw a corresponding graph. This curve is usually a convex (sublinear) curve. The less convex it is, the better. In our actual experiments, we could not calculate these curves as stated because we were unable to run the program on a single processor because of its size. Instead, we calculated the speed-up factor against the running time of 2 processors. The speed-up curves for the experiments using the CGA and the DGA are shown in Appendix 2. As can be seen, these curves are not smooth. This is due to the fact that for experiments with different number of processor, different random seeds have to be used in the different processors, resulting in a slightly different behavior of the algorithm. In the case of the CGA, the speed-up is better as the size of problem increases. The reason is that the overhead the CGA carries, due to the global selection and the synchronization delays of the master processor, is sufficient to slow down the speed-up in this small size problem. In the case of the DGA, there is no global selection and fewer synchronization delays. So the speed-up curves show less change as the size of the problem increases although it still appears that speed-up is better in larger size problem. We would like to note that as the local selection we used is a variant of Selection 2: we take 5 samples and when we overwrite a structure, we make sure that it is overwritten by a better structure, if the one to be overwritten is better, no overwriting occurs. As local improvement method we used simulated annealing [3, 4, 18, 25] where the initial temperature is chosen to be some fraction of the standard deviation of the values of the population and this temperature is decreased exponentially. Also we used a stopping mechanism which is different from the standard stopping mechanism which consists in just counting number of evaluations. A more detailed version of this DGA is shown in Appendix 3.

6. Conclusion

We have shown that genetic algorithms can be modified to become distributed asynchronous algorithms, which we called distributed genetic algorithms. This is done by localizing the selection step and distributing it together with the evaluation and recombination steps to a pool of processors. Our experiments indicate that the solutions obtained by DGAs are as good as the ones obtained by the standard genetic algorithms which we implemented as centralized genetic algorithms. The advantage of DGAs are that they are reliable, natural algorithms which, when implemented on a parallel machine, can result in very fast search algorithms. Other, more conventional, sequential search algorithm are much harder to parallelize. This point is well explained in [17]. In short, many such algorithms accumulate improvements on a single structure. This implies that one cannot easily break up those improvements into small pieces and use a pool of processors to perform them independently, i.e., it is usually the case that one piece of improvement has to be done first in order for another piece of improvement to become effective. Distributed genetic algorithms do not suffer from such complications. Despite the contention problems (which are minimal), speed-up factors can be expected to be much higher than speed-up factors obtained for other

search algorithms, for example [9]. It would be quite worthwhile to conduct an empirical study comparing the performance of DGAs with that of parallelized versions of other search algorithms.

References

[1] J. Baker," Adaptive Selection Methods for Genetic Algorithms", Proc. of an Int'l Conf. on Genetic Algorithms and Their Applications, pp. 101-111 (July 1985).

[2] BBN Advanced Computers. The Uniform System approach to programming the Butterfly parallel processor. Rep. 6149, Version 2.

[3] E. Bonomi, Jean-Luc Lutton, " The N-city Traveling Salesman Problem: Statistical Mechanics and the Metropolis Algorithm " SIAM Review Vol. 26 No. 4 October 1984 pp 551-568

[4] V. Černý, " Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm" Journal of Optimization Theory and Application Vol.45 No. 1 January 1985 pp 41-52

[5] L. Davis, "Job Shop Scheduling with Genetic Algorithms", Proc. of an Int'l Conf. on Genetic Algorithms and Their Applications, pp. 136-140 (July 1985).

[6] L. Davis, "Applying Adaptive Algorithms to Epistatic Domains", Proc. of 9th IJCAI, pp. 162-164 (Aug 1985).

[7] K.A. De Jong, "Adaptive System Design: a Genetic Approach", *IEEE Trans. Syst., and Cyber. Vol. SMC-10(9)*, pp. 556-574 (September 1980).

[8] K.A. De Jong, "Genetic Algorithms: a 10 Year Perspective", Proc. of an Int'l Conf. on Genetic Algorithms and Their Applications, pp. 169-177 (July 1985).

[9] Raphael A. Finkel, John P. Fishburn " Parallelism in Alpha-Beta Search" Artificial Intelligence 19(1) September 1982, pp 89-106

[10] J.M. Fitzpatrick, J.J. Grefenstette and D. Van Gucht, " Image Registration by Genetic Search", Proceedings of IEEE Southeastern April 1984, pp 460-464

[11] D.E. Goldberg and R. Lingle, "Alleles, Loci, and the Traveling Salesman Problem", Proc. of an Int'l Conf. on Genetic Algorithms and Their Applications, pp. 154-159 (July 1985).

[12] J. J. Grefenstette, "Optimization of Control Parameters for Genetic Algorithms", IEEE Trans. Systems, Man, and Cybernetics (1985)

[13] J.J. Grefenstette, "Parallel Adaptive Algorithm for Function Optimization" Technical Report CS-81-9 Computer Science Dept. Vanderbilt University November 1981

[14] J.J. Grefenstette, R. Gopal, B.J. Rosmaita and D. Van Gucht, "Genetic Algorithms for the Traveling Salesman Problem", Proc. of an Int'l Conf. on Genetic Algorithms and Their Applications, pp. 160-168 (July 1985).

[15] J. J. Grefenstette, "Incorporating Problem Specific Knowledge into Genetic Algorithms", To appear

[16] J. Holland, *Adaptation in Natural and Artificial Systems*, Univ. of Michigan Press, Ann Arbor (1975).

[17] P. Jog, D. Van Gucht, "Parallelization of Probabilistic Sequential Search Algorithm", Technical Report, Indiana University, April 1987 To appear in the Proc. of the 2nd Int'l Conf. on Genetic Algorithms and Their Applications.

[18] S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi, "Optimization by Simulated Annealing", *Science Vol. 220*(4598), pp. 671-680 (May 1983).

[19] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys (Ed), *The Traveling Salesman Problem*, John Wiley & Sons Ltd (1985).

[20] S. Lin and B.W. Kernighan, "An Effective Heuristic Algorithm for the Traveling Salesman Problem", *Operations Research* 1972, pp. 498-516.

[21] M. L. Maudlin, " Maintaining Diversity in Genetic Search", *Proc. of an Int'l Conf. on Genetic Algorithms and Their Applications*, pp. 247-250 (July 1985).
[22] R. Rettberg, R. Thomas "Contention is No Obstacle to Shared-Memory Multiprocessing", *CACM* December 1986, pp. 1202-1212
[23] D. Smith, "Bin Packing With Adaptive Search", Proc. of an Int'l Conference on Genetic Algorithms and Their Applications, pp. 202-206 (July 1985).
[24] S.F. Smith, "Flexible Learning of Problem Solving Heuristics Through Adaptive Search", Proc. of 8th IJCAI (Aug. 1983).
[25] J. Y. Suh, D. Van Gucht, "Incorporating Heuristic Information into Genetic Search", Technical Report, Indiana University, February 1987 To appear in the Proc. of the 2nd Int'l Conf. on Genetic Algorithms and Their Applications.

APPENDIX 1

** Illustration on Parameters and Tables of Statistics **
** On TSP Experiment. **

Population Size : the number of population on given time.
Structure Length : the length of structure (in this case, the number of cities).
Crossover Rate : the portion of population undergoing cross-over. the rest will undergo the local improvement.
Local Rate : if you mutiply this ratio to the structure length, you will get the number of local improvement attempts to be done for a structure.
Selection Rate : the probability that each strucuture will undergo the LOCAL SELECTION.

** How to read a table **

Population Size = 100
Structure Length = 100
Crossover Rate = 0.5
Selection Rate = 0.5

Algorithm		DGA		CGA	
Local Rate		5 %		10 %	
lattice	exp. 1	101.6	(139:14000)	100	(188:16910)
	exp. 2	101.6	(149:15000)	100.8	(200:17786)
	exp. 3	100.8	(179:18000)	100	(207:18865)
	exp. 4	100.8	(139:14000)	100	(237:22538)
	exp. 5	102.0	(189:19000)	100.8	(163:13650)

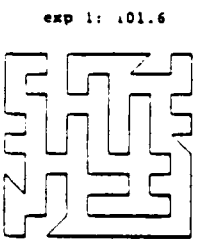
First 3 parameters above the table are those of both DGA and CGA. The last one "Selection Rate" is that of DGA. The above table is for TSP experiment for Lattice problem. We conducted 5 experiments for two different algorithms. The first row shows the result of DGA with Local rate 5 %. The description "exp. i" means experiment i. nnn.n (ggg: tttt) means that nnn.n is the best solution after ggg generations which took tttt trials. ggg and tttt for DGA are approximate values.

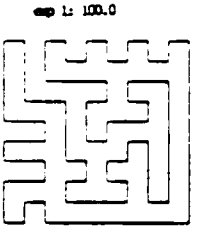
** In what follows we show the results obtained for the three TSP pro-
** blems shown in Figure 2.

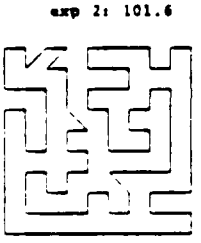
1. LATTICE problem

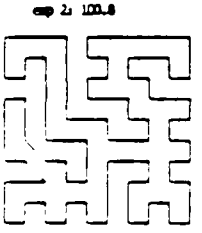
Population Size = 100
Structure Length = 100
Crossover Rate = 0.5
Selection Rate = 0.5

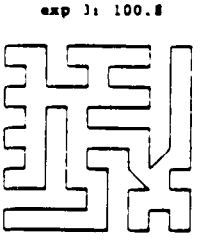
Algorithm		DGA	CGA
Local Rate		5 %	10 %
lattice	exp. 1	101.6 (139:14000)	100 (188:16910)
	exp. 2	101.6 (149:15000)	100.8 (200:17786)
	exp. 3	100.8 (179:18000)	100 (207:18865)
	exp. 4	100.8 (139:14000)	100 (237:22538)
	exp. 5	102.0 (189:19000)	100.8 (163:13650)

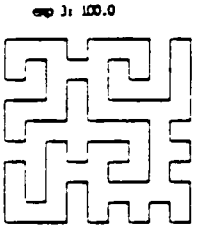


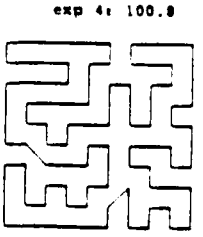


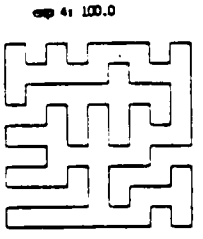


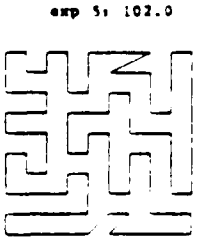


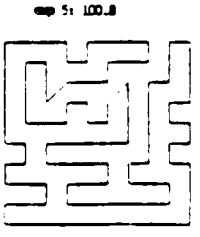








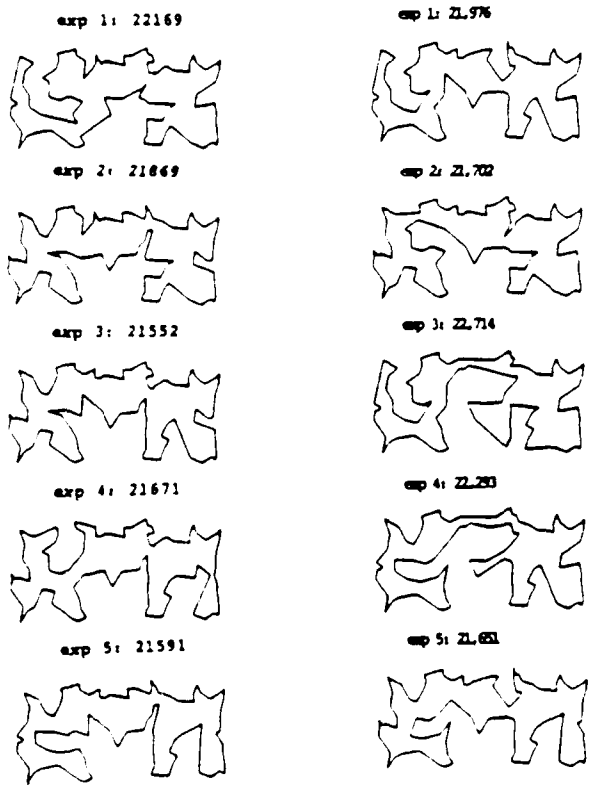




2. KROLAK problem

Population Size = 100
Structure Length = 100
Crossover Rate = 0.5
Selection Rate = 0.5

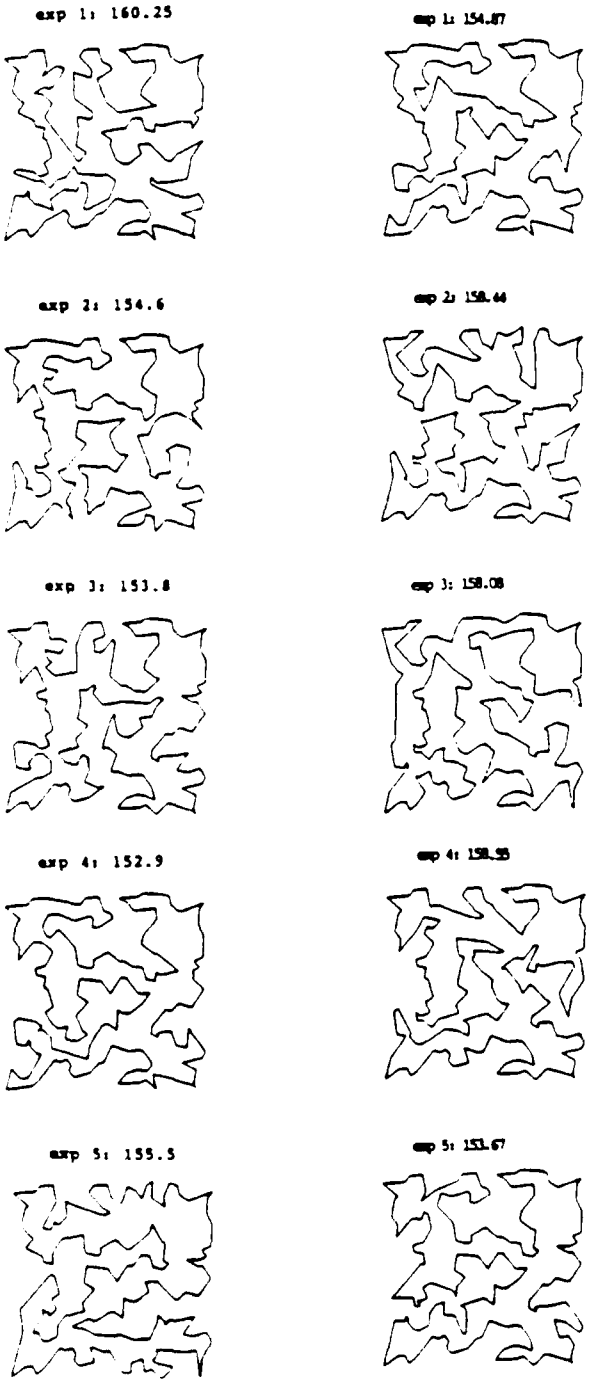
Algorithm		DGA	CGA
Local Rate		5 %	10 %
krolak	exp. 1	22169 (319:32000)	22293 (373:29435)
	exp. 2	21869 (489:49000)	22714 (386:30361)
	exp. 3	21552 (369:37000)	21702 (403:31536)
	exp. 4	21671 (259:26000)	21976 (627:49482)
	exp. 5	21591 (409:41000)	21651 (679:49745)



3. 200 cities

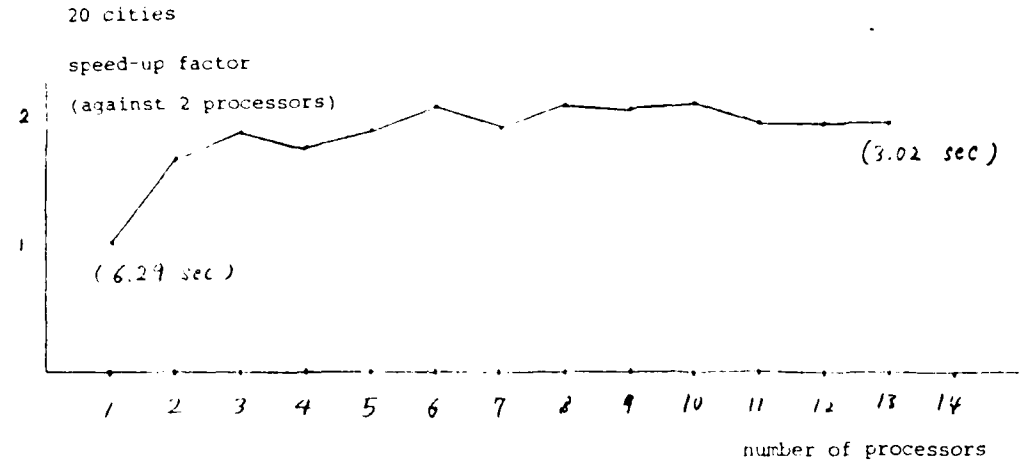
Population Size = 100
Structure Length = 200
Crossover Rate = 0.5
Selection Rate = 0.5

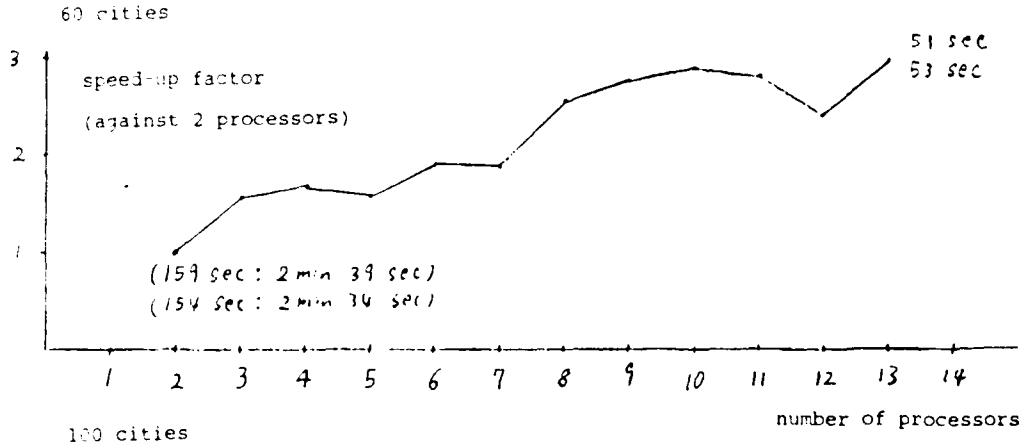
Algorithm		DGA	CGA
Local Rate		5 %	10 %
200	exp. 1	160.2 (939:94000)	154.8 (679:52312)
	exp. 2	154.6 (719:72000)	158.4 (999:78890)
	exp. 3	153.8 (919:92000)	158.0 (711:57030)
	exp. 4	152.9 (699:70000)	158.5 (768:60148)
	exp. 5	155.5 (679:68000)	153.6 (946:72553)

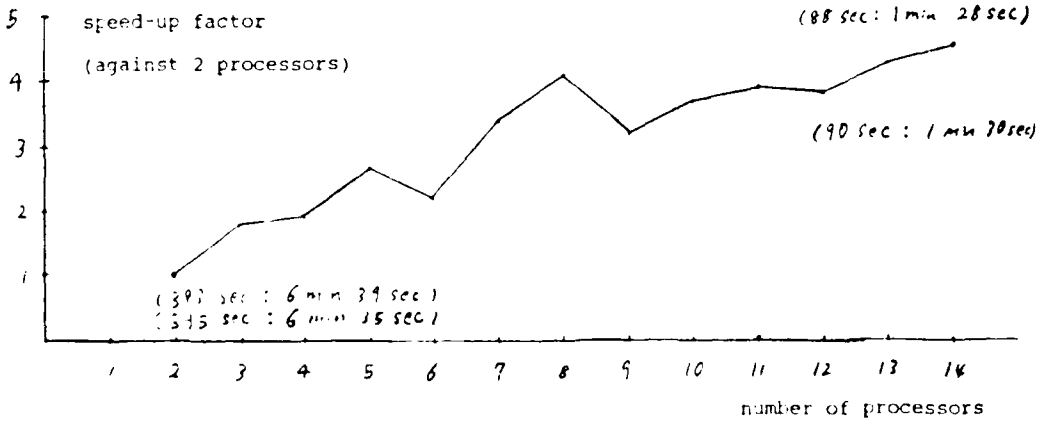


APPENDIX 2

Speed-up Curve of CGA

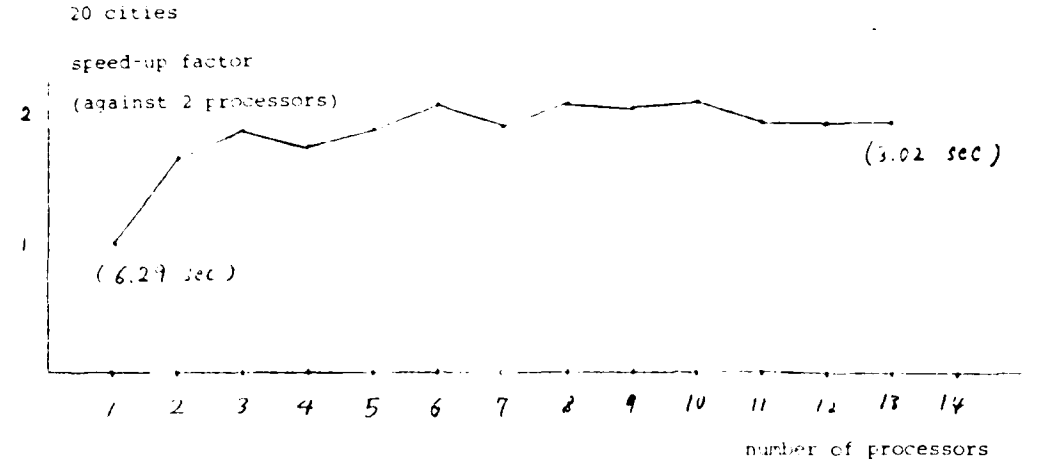


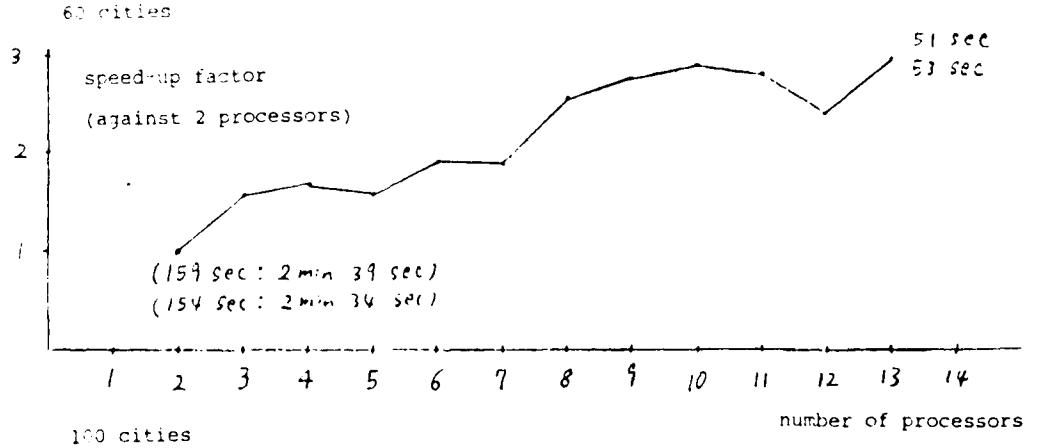


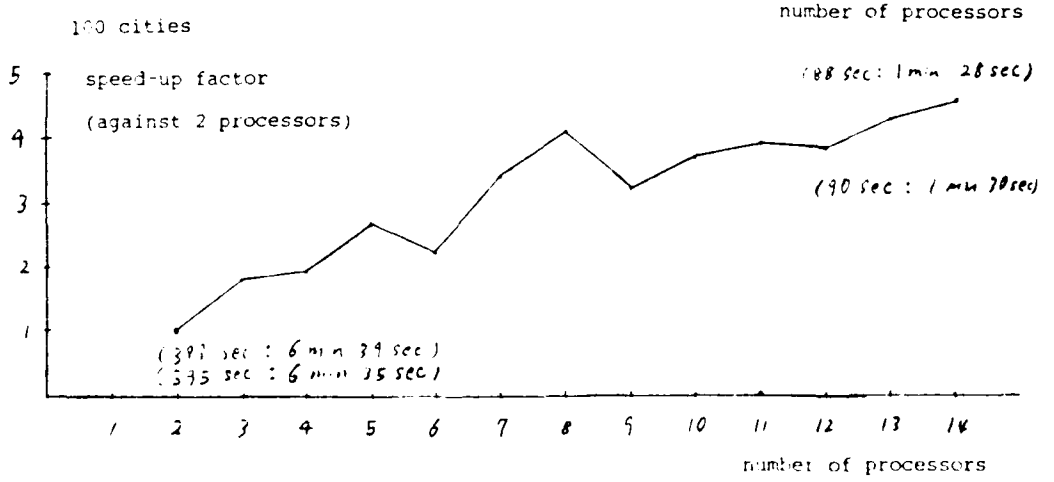


APPENDIX 2

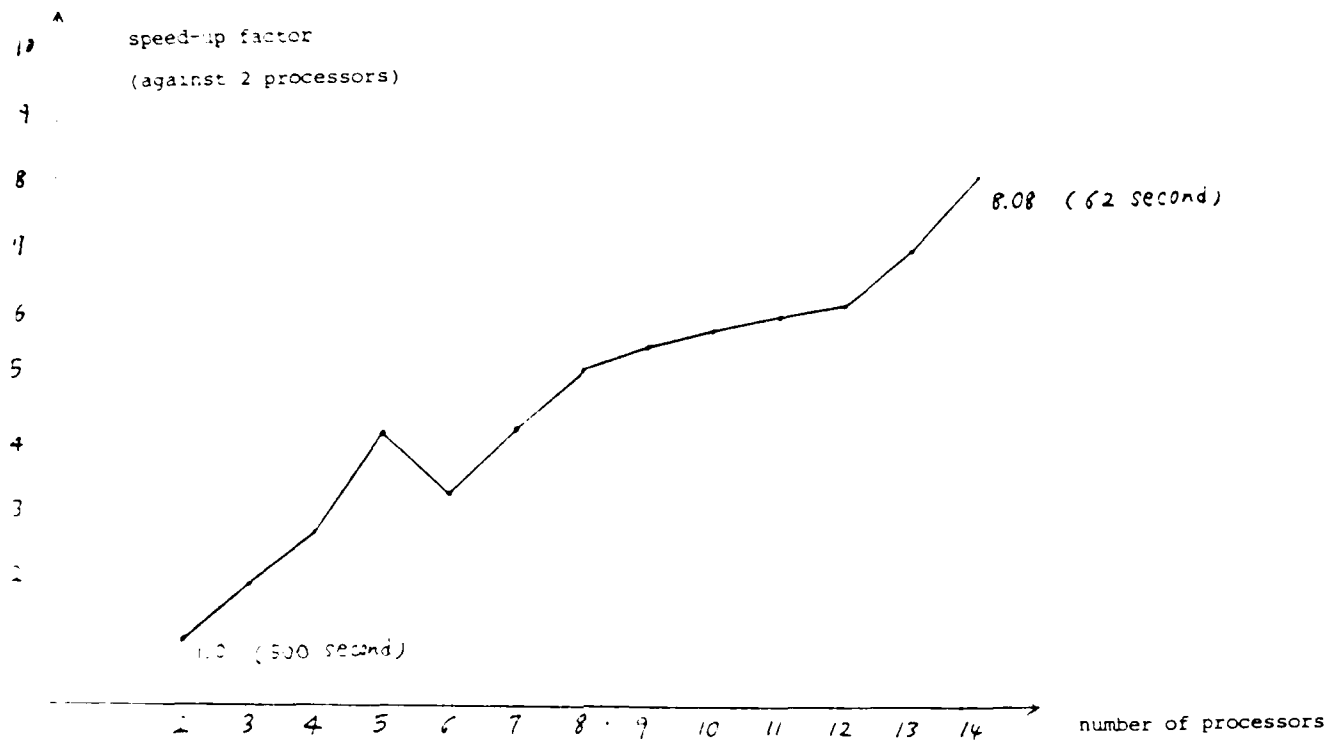
Speed-up Curve of CGA







100 cities



List of Solutions of CGA

1. 20 cities

Optimal Solution : 20
Population Size : 20
Processors used : 14

best solution	experiment 1	21.23
	experiment 2	20.00
	experiment 3	20.00
	experiment 4	21.23
	experiment 5	20.00

2. 60 cities

Optimal Solution : 60
Population Size : 60
Processors used : 14

best solution	experiment 1	60.00
	experiment 2	61.65
	experiment 3	60.82
	experiment 4	60.82
	experiment 5	61.65

3. 100 cities

Optimal Solution : 100
Population Size : 100
Processors used : 14

best solution	experiment 1	100.82
	experiment 2	100.82
	experiment 3	100.82
	experiment 4	100.00
	experiment 5	100.00

List of Solutions of DGA

1. 20 cities

Optimal Solution : 20
Population Size : 20
Processors used : 14

best solution	experiment 1	20.00
	experiment 2	20.00
	experiment 3	20.00
	experiment 4	20.00
	experiment 5	20.00

2. 60 cities

Optimal Solution : 60
Population Size : 60
Processors used : 14

best solution	experiment 1	61.99
	experiment 2	61.65
	experiment 3	61.99
	experiment 4	60.82
	experiment 5	61.41

3. 100 cities

Optimal Solution : 100
Population Size : 100
Processors used : 14

best solution	experiment 1	101.41
	experiment 2	101.65
	experiment 3	100.82
	experiment 4	100.82
	experiment 5	100.82

APPENDIX 3

Detailed Description of the DGA

Parameters

l : the number of cities.
m : the size of the population (equivalently m = number of processors)
P : the population.
$\mathbf{t} = (t_0, t_1, ..., t_{m-1})$: array of private clocks of the processors.
Each initially starts with 0.
$\mathbf{conv} = (conv_0, conv_1, ..., conv_{m-1})$: array of private counters,
used for determining the convergence of the algorithm.
Each initially starts with 0.
C : the set of all possible tours.
R^+ : the set of positive real numbers.
f : the function of C to R^+, which returns the performance measure of tour, i.e., the length of tour.
p_c, p_s, p_l: the probability that a tour will undergo cross-over, local improvement (simulated annealing), and local selection respectively.

Setting

a. There are m processors, $proc_0, proc_1, ..., proc_{m-1}$. Each processor $proc_i$ has one local tour c_i in its local memory.

b. c_i is stored in an array cur_i. There is another array new_i in the local memory which is used to hold a new tour derived by **cross-over** or **local improvement (simulated annealing)**.
There are two additional arrays, mom_i and dad_i. They are needed to avoid the problem of structures being overwritten while they are being used for other operations.

c. There are other local variables cv_i which holds $f(cur_i)$.
nv_i is the same kind of local variable which will store $f(new_i)$, if necessary.
In the same fashion, $momv_i$ and $dadv_i$ hold $f(mom_i)$ and $f(dad_i)$ respectively.

DGA()

begin

GenTaskEachProc (**Initialize**, 0); /* m processors run **Initialize** in parallel */

GenTaskForEachProc (**Operation**, 0); /* m processors run **Operation** in parallel */

/***** **OutputBest** () *****/

Output the best tour c^* and its length $f(c^*)$;

end

Initialize ()

begin

/* assume that the local memory has c_i */

Randomly generate a tour c_i and store it in cur_i; /* initialize a tour */
Compute $f(cur_i)$ and store it in cv_i; /* initialize the length of tour */
$t_i = 0$; /* initialize a local time step */
$cont_i = 0$; /* initialize a convergence counter */
end

Operation ()

begin

/* assume that the local memory has c_i */

Repeat
lock the array cur_i and its performance cv_i;
copy them into mom_i and $momv_i$;
unlock cur_i and cv_i;

with probability p_c:
begin
Randomly pick a tour c_j from cur_j in the local memory of $proc_j$;

lock the array cur_j and its performance cv_j;
copy them into dad_i and $dadv_i$;
unlock cur_j and cv_j;

Do **cross-over** with mom_i, dad_i to produce new_i;
compute $f(new_i)$ and store it into nv_i;

lock cur_i and cv_i;
swap cur_i, cv_i with new_i, nv_i; /* locally update the population */
unlock cur_i and cv_i;
end;

with probability p_s:
begin
Do **simulated annealing** with mom_i to produce new_i;

lock cur_i and cv_i;
swap cur_i, cv_i with new_i, nv_i; /* locally update the population */
unlock cur_i and cv_i;
end;

with probability p_l:
Do **local select**;

$t_i + +$;

Until (locally converged());

end

cross-over ()

The cross-over operator used here is the so-called *heuristic cross-over operator*. It is designed to ensure that the resulting configuration is a valid tour and a possibly better one. This kind of cross-over was introduced in [14]. It goes as follows:

> **note** : In the below, an edge means a line connecting
> two cities.
> Pick a random city as the starting point for the child's
> tour. Compare the two edges leaving the starting city in
> the parents and choose the shorter edge.
> Continue to extend the partial tour by choosing the shorter
> of the two edges in the parents which extend the tour. If
> the shorter parental edge would introduce a cycle into the
> partial tour, then extend the tour by a random edge. Con-
> tinue until a complete tour is generated.
> (This phrase is taken from [14])

simulated annealing ()

This operation replaces the random mutation operator used in previous genetic algorithms. As mentioned in [25], it plays a critical role in improving the efficiency of GA. It was inspired by the $2-opt$ operator of Lin and Kernighan [20]. Several researchers used it to devise an efficient algorithm for the TSP. Especially some of them saw it as a good way of generating a new tour from the existing one in the application of simulated annealing to the TSP [3,4,18]. Ours is a modified version of the version used by Černý [4] and basically is the following:

> Pick randomly 2 edges e_0, e_1 where e_0
> is from city x_0 to y_0 and e_1, from x_1 to y_1.
> Make sure that all 4 cities are different.
> Let d_0 be an edge from x_0 to y_1 and d_1, from x_1 to y_0.
> Let the new tour τ be the modification of the old one where edges
> d_0, d_1 replace e_0, e_1 respectively and those edges between
> e_0 and e_1 are reversed.
> Let $\Delta e = |e_0| + |e_1|$,
> $\Delta d = |d_0| + |d_1|$
> As can easily be seen,
> the length of τ = (the length of old tour) + $\Delta d - \Delta e$;
>
> If $(\Delta d - \Delta e < 0)$ /* new tour is shorter */
> accept τ
> Else
> with probability $\exp((\Delta e - \Delta d)/T)$
> accept τ where /* new tour is not shorter, but still acceptable */
> $T = T_0 \rho^{t_i}$,
> T_0: initial temperature,
> ρ: cooling ratio, $(0 \leq \rho \leq 1)$
> t_i: the local time step;
> (T_0, ρ) is called **annealing schedule**
>
> otherwise keep the old one.
>
> Repeat the above operations a specified amount of times
> /* This number of repetition should be specified by an user */

```
local select ()
begin
  /* assume that c_i is in the local memory */
  Randomly pick c_j in proc_j;
  if momv_i is better than cv_j then
  begin
    Randomly pick j_0, ..., j_4;
    μ = Σ_{k=0}^{4} cv_{j_k}/5;          /* rough estimate of mean of population performance */
    if momv_i is better than μ then   /* probably c_i is good enough */
    begin
      lock cur_j and cv_j;
      cur_j := dad_i;                 /* overwrite c_j */
      cv_j := dadv_i;
      unlock cur_j and cv_j;
    end
  end
end
```

Note on local select

In this routine, good tours attempt to overwrite bad ones. In this way good tours are propagated and bad ones are eliminated. To achieve this, we need to ensure that a good tour is actually a good tour in the global context. That is, it should be good among those in the whole population. The criteria for a good tour is to be above the average tour length. Since it is out of the question to compute the population average (this would defeat the gains obtained by adopting local selection rather than global selection), we use, in this case, a sample average of five tours as an alternative. This is clearly the approximation of the average tour length. If, now, a structure is better than this sample average, we are almost certainly guaranteed that it is a good structure.

```
locally converged ()
begin
  /* ε is an external constant which is quite small */
    Randomly pick j_0, ..., j_4;
    μ = Σ_{k=0}^{4} cv_{j_k}/5;
  /* does the population appear to be converged ? */
  if |μ − cv_i| < ε   conv_i + +;
  if conv_i > t_i/10 return(true);
    else return(false);
end
```

Rationale for using this stopping algorithm

In the CGA, the average performance is smoothly going down and the variance of population performance is almost monotonically decreasing. The population eventually converges and the time to get to the converged population stays more or less constant for a given instance of problem. In contrast, the DGA shows a more noisy behavior. The average performance does go down and the variance shrink, but they tend to fluctuate much more than those of the CGA. It is hardly exepected that the variance of performance

of population converges to 0, and even if it does, it takes an unreasonably long time. It is usually the case that the best structure is found well before the convergence occurs. There is another problem with the convergence pattern of DGA. Many times the population looks converged and no further improvement of population appears possible, only to diverge and a better structure emerges. That is, DGA often shows "false convergence". Thus it is not reasonable to use the convergence of the population as a criterion for stopping the DGA. We need to look for another alternative. It should be the one which reflects in some way the degree of convergence but is not so rigid as to base everything on the convergence of the population itself. So we adopt the following approach: In each generation, we pick the sample of small size and compute its average performance. If the performance of current structure is close to this average, we increment a counter by 1 (The closeness is decided, for example, by whether they are within a few percent of standard deviation of the initial population). As the counter increases, it shows that the population is converging. If the counter is eventually goes beyond some fraction (in our case, fraction is 10 percent) of generations taken so far, the processor signals that it reached a converged state. If all processors reach this state, the DGA stops.
The advantage of this routine is as follows:

1) It works in noisy environments: It does not matter if the variance fluctuates or not. As long as the instance of close convergence far outnumbers the other, the DGA will stop.

2) It does not stop due to false convergence: When theDGA enters a false convergence state, it does not stop there but goes until the instances of converged states in the processors occurs enough times, does enabling the population to diverge again and produce better structures.

3) It may serve as an general stopping routine since it is problem and problem size independent.

Some of Actual Data Structures in Butterfly Implementation

Here, $CurPop$, $Neu\,Pop$, $CurF$ and $Neu\,F$ are implemented as arrays of pointers which points to cur_i, neu_i, cv_i, nv_i respectively. That is,

$*CurPop[i] = cur_i$
$*Neu\,Pop[i] = neu_i$
$*CurF[i] = cv_i$
$*Neu\,F[i] = nv_i$.

Each processor has its local copy of these four arrays. This is again for avoiding contention. In the case of CGA, these pointers are updated synchronously after **Operation** is done. Since each structure is updated asynchronously in DGA, pointers to its storage and to its performance are updated at different times. By knowing the location where a pointer is stored instead of that pointer itself, we can update them locally and still other processor can access correct pointers. Pointers itself can change but the location of their storage remains unchanged.

A Software Tool For Building Supercomputer Applications.

Dennis Gannon
Daya Atapattu
Mann Ho Lee
Bruce Shei

Department of Computer Science
Indiana University
Bloomington, Indiana

ABSTRACT

We describe a software tool that consists of an interactive environment for helping users restructure programs to optimize execution on parallel/vector multiprocessors. The system is used to help programmers fine tune codes that have already been passed through an automatic parallelizing system or codes have been designed from the start from new parallel algorithms. In particular, programs optimized for one machine can be easily reoptimized for another using this system. To accomplish this task, the tool provides mechanisms to give the programmer feedback concerning the potential performance of his code on the chosen target machine and allows the user an interactive means to guide the system through a sequence of automatic program transformations.

1 INTRODUCTION

Five years ago the subject of Parallel Computation was an exotic sub-field of computer science that consisted largely of theoretical studies of potential parallel algorithm performance, a few university hardware projects, even fewer software design efforts and (with the exception of Denelcor) no industrial products. Five years later, we now find that all U.S. supercomputers, and over one dozen mini-supercomputer vendors have entered the market with some form of general purpose parallel processing system. It is expected that every computer company will offer a scalable multiprocessor by the end of the decade.

Unfortunately, very few of these systems provide a software environment for building parallel programs that goes beyond a standard sequential language compiler and a microtasking library to support parallelism. None of our most widely used programming languages (C, Fortran, Lisp) have been given official extension to support concurrency and no two vendors agree on any of the unofficial extensions. The task of porting parallel codes from one machine to another now involves large amounts of recoding to make a program work, and large amounts of restructuring of the algorithm organization to make it work well.

The primary concern of the vast majority of users is to be able to exploit the power of a supercomputer without sacrificing portability. Consequently, these users want an automatic system such as those provided by Pacific-Sierra or KAI which will provide substantial improvement in a code without any effort by the programmer.

On the other hand, there are still a large number of hardy users of these new systems who are not content to live with the results of an automatic vectorizer or parallelizer. These are the users who are willing to invest a "reasonable" amount of extra effort if it might mean a doubling of performance beyond what the automatic system delivers. The key to this extra performance is the cost. How much is a "reasonable" amount of extra work?

In this paper we describe a programming tool designed to help users of parallel supercomputers retarget and optimize application codes. In a sense, the system can be viewed as a tool to help users "fine-tune" the output of an automatic system or, if he or she has been so inspired, optimize the design of a new parallel algorithm.

The system is an interactive program editing and transformation system that helps the user with this task. Each program that enters the system is completely parsed and all data dependences are recorded. The user then works with the system to restructure his code to a form suitable for a given target architecture. If the target is known to the system, it monitors the users transformations to the code. If the user attempts to transform the program in violation of the original semantics of the code he is warned that a change in the meaning of his program has taken place. At any time the user can ask the system to tell him what legal parallelizing transformation can be applied to a segment of selected code. More important, he can ask the system to make the program modifications the user desires. In this mode, the user is assured of the correctness of the changes in his code.

In its current form, the system, known as the Blaze Editor or "Bled", can support either FORTRAN (with Cedar and Alliant 8x extensions) or Blaze (a Pascal based functional language designed by Mehrotra and Van Rosendale [14]). In the future we plan to support C, C++ and Cedar Parallel C [9]. The target machines supported currently include the BBN Butterfly, the Alliant FX/8 and the Cedar System [7].

This tool is one part of a much larger programming environment known as the Faust Project. This effort, based at the Center for Supercomputer Research and Development in Urbana, Illinois has designed a common software platform for a number of programming tools including a performance analysis package, a program debugger, Bled, and a graphics based program maintenance system. All the software has been written to use the X system from the MIT project, Athena and, therefore, it will run on any Unix-based workstation.

In this paper we describe the current Bled system as well as two important extensions that are being added at the time of this writing.

One extension is a performance prediction package that can be invoked from within Bled to help the user choose which formulation of his algorithm will run best. There are two components to this performance prediction package. First is a code generation estimator that can give the user feedback in the form of estimates of such quantities as the ratio of vector instructions to data movement, or the amount of code devoted to synchronization overhead. Second, this package provides estimates of cache behavior and local memory utilization.

The second extension is a portable runtime environment that supports a dynamic "microtasking" facility that incorporates ideas from the Argonne Schedule package [5] and the MIT multilisp system [10].

2 A SAMPLE USER SCENARIO

To illustrate how a user would interact with this system we shall step through a very simple example. The Blaze subroutine below is a simple matrix times vector routine.

```
Procedure MatVec(n,A,x) returns: y;
param A: array[1..n, 1..n] of real;
    x,y: array[1..n] of real;
    n: integer;
begin
     for j in 1..n loop
          for i in 1..n loop
               y[i] := y[i]+A[i,j]*x[j];
          end;
     end;
end;
```

The user loads this program into the system as if he were entering a text editor. The result is a window displaying the program and a list of menu headers (as illustrated in Figure 2.1). The first thing he may wish to do is to tell the system which machine is the intended target of this optimization. Currently this menu only lists three active choices: the BBN butterfly, the Alliant FX/8 and the Illinois Cedar. (We plan to extend this list to include the IBM RP3, the CRAY 2, the Connection Machine, and the ETA-10 during the next two years.)

The significance of having the user tell the system about the choice of target is threefold. First, and most obvious, we would like to have the system generate code for the given target. We will say more about this later. Second, it is important that program transformations that are inappropriate for the target machine be disabled. Third, and of most immediate concern to the user, selecting a target will enable the appropriate performance estimators which are described below.

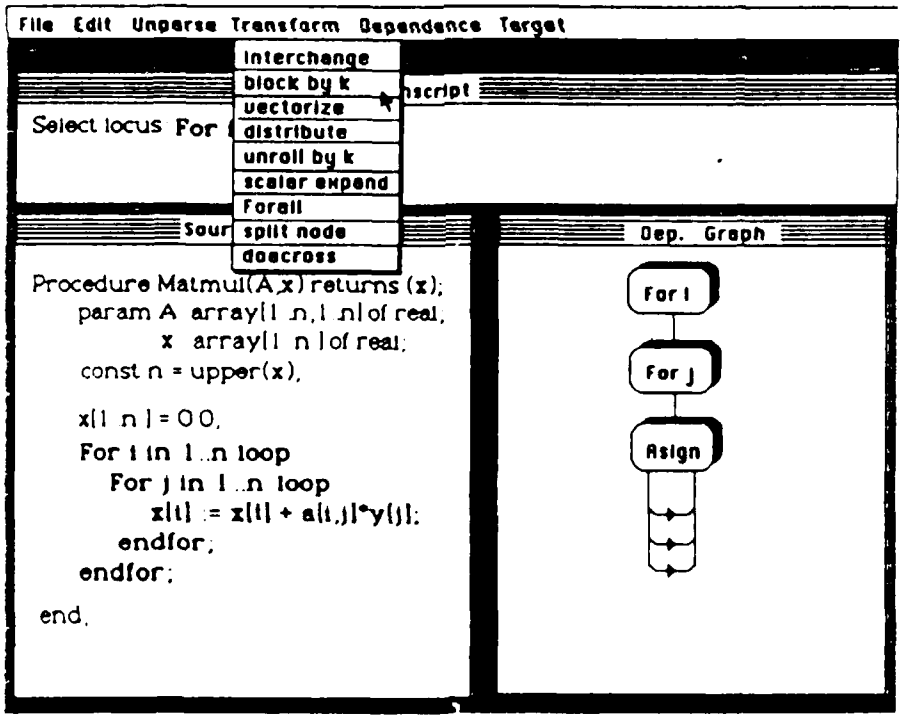


Figure 2.1. BLED Screen After Program Load.
The Window on the Right is the Data Dependence Graph.

To begin working with the system the user selects a segment of code, which we call a 'focus', on which to apply the tools BLED provides. For example, suppose the target is the BBN Butterfly and the user picks the innermost loop (by a mouse selection) as his focus. Among the menu headings he has at the top of the screen, is one called "Transforms". This menu contains a list of program restructuring transformations that can be used to expose concurrency or make parallelism explicit. For example, the user may have decided that he wishes the innermost "for" loop to be run in parallel. By selecting the transformation "forall" the system will first verify that the transformation can be legally applied. This requires a search of the data dependence graph to make sure that the appropriate conditions are satisfied so that the transformation can be legally applied. (A more detailed discussion of this process is given in Section 3.) In this case the transformation is legal and the code now takes the form

```
for j in 1..n loop
     forall i in 1..n do
          y[i] := y[i]+A[i,j]*x[j];
     end;
end;
```

Following this operation, the user could invoke the code generator, but a better use of the system is to first invoke the analysis tool. This involves the selection of a menu item "analysis: Parallel Loop". Because the target machine, the Butterfly, executes such loops at the cost of a function call and an atomic increment to the index, the reply will be in the form

```
Loop Overhead to Body ratio > 50%.
Suggestion: increase granularity by blocking,
     merging or loop interchange.
```

Of the three suggestions, loop interchange is the easiest. After this operation the loop takes the form

```
forall i in 1..n do
     for j in 1..n loop
          y[i] := y[i]+A[i,j]*x[j];
     end;
end;
```

The "forall" loop body will now be large with respect to the loop overhead, but another problem that may inhibit performance is memory contention for shared data. In this case a second analysis tool called "analysis: Cache Management" can be invoked. As will be described in more detail in section 4, this tool will report the following information when the program focus is the "forall" loop.

```
Cache/Local memory analysis for iterate i:
     Suggest local copies of: A[i,1..n], x[1..n]
     for n=100 hit ratios will be 0.99, 0.9999
```

This (too cryptic) message suggests that local copies of these variables be made in each processor. A major shortcoming of this form of the analysis is that the user is not informed of the penalty for failing to take this advise. In section 4 we discuss the future extensions of this analysis that will provide expected improvements in multiprocessing efficiency as a

result of this memory management. The resulting program will take the form

```
forall i in 1..n do
var: x_local, A_local: array[1..n] of real;
     A_local[1..n] := a[i,1..n];
     x_local[1..n] := x[1..n];
     for j in 1..n loop
          y[i] := y[i]+A_local[j]*x_local[j];
     end;
end;
```

Clearly *x_local* need only be initialized once per processor but in Blaze we have no way to express this, so this task is left to the code generation step.

Optimizing Code For The Alliant FX/8

Suppose the target machine were chosen to be the Alliant FX/8. Each processor on this machine has a vector instruction set and a vector register set that generalizes and extends the M68020 which forms the basis of the rest of the processor design. The execution of parallel loops requires almost no overhead, but best performance is achieved only if the shared cache memory and the vector instruction set is optimally utilized.

To optimize the matrix-vector routine for the FX/8 we would begin by recognizing that the vector instructions operate from registers of length 32. For this reason, the user may wish to "block" the inner loop in vector segments of length 32. This requires two steps. First the user selects "block by 32" from the transformation menu as illustrated by the screen dumps in Figures 2.1 and 2.2. In the second step the inner loops are vectorized.

The result of these two operations is shown below.

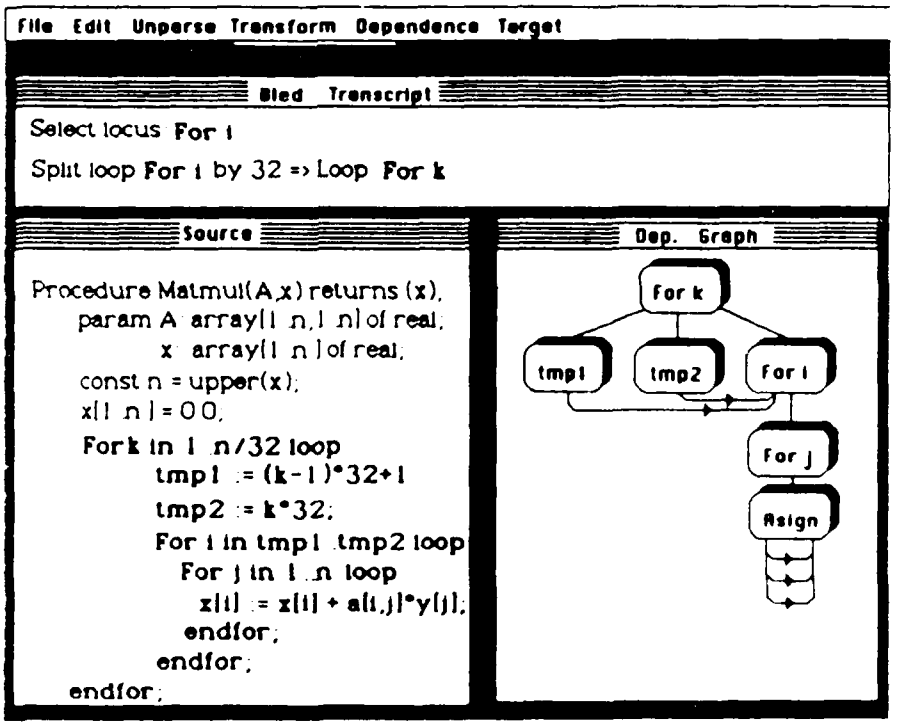


Figure 2.2. System State After Loop Blocking Transformation.

```
for j in 1..n loop
    for k in 0..n/32-1 loop
        k1 := 32*k+1;
        k2 := 32*(k+1);
        y[k1:k2] := y[k1:k2] + A[k1:k2,j]*x[j];
    end;
end;
```

At this point, a third component of the performance analyzer can be invoked. The **Vector Code Analyzer** will make an estimate of the quality of the vector instruction utilization for this loop. In this case it will report

```
Vector instructions   Frequency
Load/Store            2*n*n/32
Multply-Add           1*n*n/32
Arith. Efficiency = 33% (for n=100)
```

The vector efficiency term is computed based on the fact that for diadic vector instructions the processor is capable of two floating point operations every clock cycle. Vector loads and stores do not contribute to the total number of arithmetic operations but they do consume large amounts of time. If we look at the inner loop we notice that the indecies on the vector segments for y do not depend upon j. In this case one simple "loop interchange" transformation will bring the j loop inside and the processor need only load and store each y segment once. The rest of the time it can remain in a vector register. Parallelizing the outer loop gives the code below and the corresponding statistics.

```
forall k in 0..n/32-1 do
    k1 := 32*k+1;
    k2 := 32*(k+1);
    for j in 1..n loop
        y[k1:k2] := y[k1:k2] + A[k1:k2,j]*x[j];
    end;
end;
```

The output from the code analyzer is now

```
Vector instructions   Frequency
Load/Store            2*n/32
Multiply-Add          1*n*n/32
Arith. Efficiency = 99% (for n=100)
```

Notice that this simple transformation has had the effect of a three fold improvement in vector unit efficiency. The resulting code is, in fact, the fastest matrix-vector form for this machine.

3 SYSTEM ORGANIZATION AND THE TRANSFORMATION MODULES.

The system is organized as a set of four interacting modules: the parser/analyzer, the transformation module, the performance estimator, the code generator, and the user interface manager. In this section we describe the Parser/analyzer, the user interface and the program transformation modules.

The Program Parser and Analyzer

At the time a program module is loaded into the system it is completely parsed and a flow control dependence graph and symbol table is constructed. The control dependence

graph is based on a statement level version of the PDG of Feranti and Ottenstein [6]. Each program statement is represented by a graph node in the control dependence graph. An arc goes from one node to another if the latter is "control dependent" (in the sense of [6]) on the former. In the case of structured programs this graph is always a tree as illustrated in Fig 3.1.

The nodes of the control dependence graph, called BIF nodes, represent program statements, declarations and control forms. Of course, this is not enough detail to describe a program. Within each program statement there are one or more expressions describing the parameters of the statement. For example, a "for loop" in Blaze or FORTRAN takes the form

For <var> from <lower_bound> to <upper_bound> by <increment>.

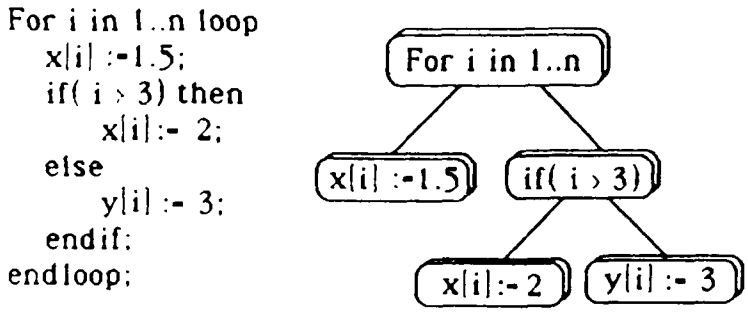


Figure 3.1. Program Control Dependence Graph

Each such expression is represented by a low-level dataflow graph which is attached to the BIF node which forms a template for the statement. While there is nothing new about this type of representation, we have found that its simplicity provides for great generality in the language it supports. Hence, the same graph can be used to represent several different programming languages.

The basic program graph has been defined with a general imperative programming language in mind. Using the same internal structure we currently represent either Blaze or FORTRAN programs. The extensions to support C are not complete, but only require the addition of a pointer type and the corresponding dereferencing operators. The way this works is that common control constructs, i.e. "for" loops, if-then-else blocks, case statements, variable assignments and function calls, form the core of the common semantics of each of these languages. There is one node type for each of these fundamental forms. For those features that exist in one language but not the others, we add extra node types. For example, FORTRAN has a "goto" statement and Blaze does not; also Blaze and C have record type variables and FORTRAN does not. The most important point is that, at the level of program semantics, and, especially in the area of control structures, there is very little difference between these languages. This means that we can write general program transformation modules that work at the level of common semantics.

We provide a special parser and "unparser" for each language. The unparser is the devise for recovering the original source for display in the text window of the system. This works by simply traversing the control dependence graph and, using the symbol table generated by the parse, reproducing the original source up to, but not including the programmer's use of white space. (We do save comments and try our best to put them back in the original positions. This can be difficult if the "original position" no longer exists after

a transformation step.) By labeling each graph with the source language type we know how to resolve any minor differences at the control structure level.

It is important to note that this syntactic independent internal representation of the language does not give us the ability to parse a FORTRAN program and unparse it as a Blaze program. Rather, it gives us the ability to design program transformation and analysis tools that are relatively independent of the syntax of the original program. The most important shortcoming of this program dependence graph is that it is limited to representing simple imperative languages. Higher order constructs such as Object and Class structures or first class functions and continuations have not been addressed.

The data dependence analyzer adds data flow information edges to the Control Dependence Graph. These take the form of distance vectors representing flow, anti, output or input dependences in the form described in [4] and [15]. In addition we have added special information about the structure of uniformly generated dependences so that cache performance modeling can be done. This is described in more detail in section 4.

The data dependence analyzer is invoked each time the program is modified by transformation or special dependence information is needed by another unit of the system. To keep this from creating excessive computational overhead the dependence analyzer works in an incremental manner. In the case of structured programs where the control graph is a tree, this works in a manner similar to the Cornell Program Synthesizer which uses an attributed grammer to build an internal representation of a program.

The User Interface And Program Transformation Modules.

The style of user interaction is identical to that of most modern software tools. The user is given a "work space" into which the program under study is loaded. By means of menu selection, the user is also equipped with a large palette of tools that he can apply to selected parts of his program. They take the form of program editing, transformation and evaluation tools.

Because the system is built on top of a solid library of graphics, menu and text manipulation routines provided by the FAUST environment, the user interface is the simplest part of the system. It consists of a loop which responds to user generated events. Events are interpreted as either a resetting of the program focus or as a call to invoke one of the tools and apply it to the current focus.

The program transformation module is organized as a collection of routines that each implement one of the transformation theorems. (These are given in various places in the literature, see [19], [17], [11], [13], [1], [2], [4], [15], [18]). Each routine first verifies that all the conditions are satisfied to guarantee that the transformation is correct, then it carries out the transformation, updates the dependence graph and redisplays the text on the screen. For example, to interchange two nested loops in a program segment we must invoke the theorem

> **LOOP INTERCHANGE: Let L_i and L_{i+1} be a perfectly nested pair of loops with $L_{i+1} \subset L_i$. The two loops can be interchanged if and only if doing so will not violate a data dependence constraint associated with any statement or pair of statements nested within L_{i+1}.**

The data dependence constraint translates into a simple mathematical condition on the distance vectors associated with the dependence. (In particular, if interchanging the i^{th} and $(i+1)^{th}$ component of the distance vector causes the vector to have a negative leading nonzero term, then the constraint will be violated.) The loop interchange module first checks to see if the current focus is rooted by a loop and that the next level down is a loop perfectly nested within the first. The dependence test is then made. If the test is passed the operation is carried out. If the test fails, the user is notified as to why the transformation could not be applied.

The user is free to override the system and to insist that the transformation be done. There are two reason for allowing this. First, the data dependence analysis may not be able to completely decide if a dependence is real. (For example, subscripts in arrays that involve function calls). In cases where it cannot decide if a dependence exists, the system takes a conservative approach and assumes the dependence is there. The programmer may have additional information that the parser and dependence analyzer does not have. For example, knowing that the function call in the index expression is of a special form that would rule out the existence of a dependence. In this case it may be safe to complete the transformation. The second reason for overriding the system is that the user may, in fact, wish to change the meaning of the program.

4 PERFORMANCE ESTIMATION

The performance module consists of two main components: The cache/local memory modeling package and the vector code generation estimator. It is important to understand that this is not a performance evaluation package that gives a detailed analysis of program behavior based on actual execution statistics. Rather, it is a tool that makes *a priori* estimates of *potential* program execution behavior. The objective is to provide the programmer some immediate feedback about the suitability of his algorithm constructs and potential problems he may encounter during execution as the code is being designed. Once the code is running on the target, the programmer will probably switch to a performance evaluation package to do final fine tuning. For the most part, this component of the system is far from complete and only a simple prototype is currently running.

The Cache/Local Memory Modeling Package

The Alliant FX/8 has a unique memory organization. All 8 CEs are connected by a crossbar switch to a shared cache. The bus bandwidth between cache and main memory is approximately one half of the total bandwidth between the cache and processors. While this provides a simple solution to the cache coherency problem, it does have major implications in the way parallel programs are organized. In particular, if one processor is using a large amount of data that is not used by the other processors, it can cause the cache to be filled at the expense of the performance of the other processors. Consequently, it is best to organize the code so that processors can share cached data if possible.

For the Butterfly machine a different problem dominates system performance. In this machine every word of memory can be reached by every processor. However, the memory that is local to a processor is significantly faster to access than that part of the address map that is nonlocal. A related issue is that of memory "hotspot" contention. This refers to the degradation problem caused by a large number of serial accesses to the same data item by a large number of processors (see [16]). It has been shown that by careful use of local memory, many of these problems can be avoided (see [3]).

Both the cache problems of the Alliant and the local memory problems associated with the BBN Butterfly can be treated by using a new theory of memory management based on dependence analysis that we have designed [8]. The key idea behind this work is that each data dependence is described by a *cache window*, which is the set of data values which when kept in cache after being referenced by the head of the dependence, will result in a cache hit after the tail of the dependence. It is shown in [8] that an algebraic model can be given to the structure of these cache windows. The model is powerful enough to allow us to give a first order estimate of both cache and local memory behavior and tell us how to optimize the program to improve memory hierarchy use.

Vector Instruction Efficiency Estimator

Like the multiprocessors from Cray and ETA, the Alliant makes extensive use of a vector instruction set. Our experience has shown that one of the first things a programmer wants to know when he is optimizing code is how well his FORTRAN 8X was translated

into vector and parallel code. In many cases he can decide to restructure a program if, for example, he realizes that the current form does not permit the code generator to use any diadic operations. On the Alliant, this type of reorganization can provide a 30% improvement in performance.

One of the estimation tools that we are adding to the system works by a pass over the selected program focus and makes a simple estimate of scalar code and loop overhead, as well as an estimate of how well the Alliant Code generator will be able to generate vector code.

This tool is related to another tool that we are adding to the Faust Workbench. We are building a code analysis package that will examine the output of the compiler and give statistics about the density of vector operations and the frequency of use of parallel constructs. Of particular concern in both tools is the effective use of diadic operation and a measure of how much time is spent moving data in and out of registers.

5 GENERATING TARGET SYSTEM CODE

Once a program has been optimized for a given architecture, a programmer will want to have it run. This means that code will need to be generated for the target. Our approach to this problem follows the tradition of other restructuring systems. Rather than generate object code, we output source code that utilizes the language extensions and special function calls that are specific to that machine. The advantage is that we can take advantage of the native code generators and optimizers that exist for that machine.

In the case of FORTRAN, we generate Alliant vector-concurrent FORTRAN 8X for that machine. For Blaze programs we generate C code for the Alliant. In the case of the Butterfly we generate a C program that utilizes the BBN "uniform system" runtime environment for both FORTRAN and Blaze.

The first problem that one confronts in mapping the computational model defined by a parallel programming language like Cedar Fortran or Blaze to a target like the Alliant or the Butterfly is the following:

> **How does one provide efficient parallel execution of a language that is historically rooted in sequential stack based semantics?**

The first instance of where this problem arises is in the execution of concurrent loops where the body of the loop contains references to names defined outside the loop. We would like to have a runtime stack which, at the point of a parallel loop invocation, can branch so that each processor has a private stack branch, but they share the part of the stack before the parallel execution call.

With the Alliant FX/8 the solution to this problem is built into the hardware. There are special instructions to set up this type of "cactus stack" and have each processor start execution at the appropriate place and time.

Unfortunately, the Butterfly presents a different computational model. Using the "uniform system" on that machine each processor gets a copy of the C program static data and has its own stack in private memory space. All shared data must be explicitly allocated in global memory. Consequently, if one processor updates a static variable or pushes an item on the stack it is invisible to all other processors. The solution to the problem for the Butterfly is to make the code generator allocate a copy of the activation record in global memory that is reachable by all processors. Any reference to data outside the scope of the loop is made through this record.

It would seem that the Alliant solution is by far superior, but there are other problems when one tries to extend the semantics in directions suggested by Schedule [5] and other portable concurrency packages. Dongarra and Sorenson have argued that parallel programmers should have the ability to write code that permits tasks to be generated dynamically and scheduled for execution when the appropriate data is available. It is important that

these tasks be "light weight", i.e. unlike a Unix process, the creation and scheduling of a task should involve no more overhead that a typical function call.

The ability to do this was supported directly by the hardware of the Denelcor HEP, but is missing from most of the current designs. The logical extension of the light weight task idea is the *future* used by Halstead in MultiLisp [10]. The principle concept is that any function call

$$x = Foo(a,b,c,d)$$

can be replaced by

$$x = future(Foo, a,b,c,d)$$

The use of *future* causes a task to be created and start executing when the appropriate value for the parameters are known. Meanwhile, the calling task continues to execute until the value of x (or any other value computed by *Foo* and assigned by side effect) is needed. At that time the caller must be suspended until *Foo* terminates.

The key problem is that suspending the caller cannot keep a processor tied up or a deadlock situation will result. Consequently, the system must encapsulate a state for the caller. Because we must assume that any processor is able to continue the execution of the caller, we cannot keep this state on the private stack of any one processor. Consequently, the task state must be saved in the global heap.

We are currently extending the code generator to support this *future* construct. There are a number of problems that we are trying to solve while completing this task.

1. How do we generate the synchronization mechanisms needed to schedule the futures?
2. How can we find an efficient implementation for our list of target machines?
3. How do we keep the implementation of *futures* consistent with the execution model for parallel loops?
4. Can we detect when a function is a good candidate for execution by futures rather than a straight sequential call?
5. In what way can we use this execution model to help us implement object oriented programming?

6 REFERENCES

1. J.R. Allen, "Dependence Analysis for Subscripted Variables and Its Application to Program Transformations," Ph.D. Thesis, Rice University, Houston, Texas, April 1983.

2. J. Allen, and K. Kennedy, "A Parallel Programming Environment," Technical Report, Rice COMP TR84-3, July 1984.

3. W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, T. Blackadar, "Performance Measurements on a 128-node Butterfly Parallel Processor," Proceedings of 1985 International Conference on Parallel Processing, pp. 531-540, 1985.

4. R. Cytron, "Compile-time Scheduling and Optimization for Asynchronous Machines," Ph.D. Thesis, University of Illinois, Urbana-Champaign, Aug., 1984.

5. J. Dongarra, D. Sorensen, "SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs," in **Characteristics of Parallel Algorithms,** Jameson, Gannon, Douglas, eds. MIT Press, 1987, pp.363-394..

6. J. Ferante, K. Ottenstein, J. Warren, "The Program Dependence Graph and Its Uses in Optimization," IBM Technical Report RC 10208, Aug. 1983

7. D. Gajski, D. Kuck, D. Lawrie, A. Sameh, "Cedar - A large Scale Multiprocessor", Proc. of the 1983 International Conference on Parallel Processing, IEEE, Aug. 1983.

8. D. Gannon, W. Jalby, "Strategies for Cache and Local Memory Management by Global Program Transformation," Proc. of 1987 International Conference on Supercomputing, Athens, Greece, June 1987, Springer-Verlag Lecture Notes in Computer Science.

9. V. Guarna, "VPC - A Proposal for a Vector Parallel C Programming Language," June 1987, Center for Supercomputer Research and Development, University of Illinois, Urbana, Illinois. Technical Report No. 666.

10. R. Halstead, "Implementation of Multilisp: Lisp on a Multiprocessor," Proc. 1984 ACM Symposium on LISP and Functional Programming, pp.25-45, Aug. 1984.

11. K. Kennedy, "Automatic Translation of Fortran Programs to Vector Form," Rice Technical Report 476-029-4, Rice University, October 1980.

12. J. Kowalik, **Parallel MIMD Computation: Hep Supercomputer and Its Applications,** The MIT Press, 1985.

13. D. J. Kuck, R. H. Kuhn, B. Leasure, D. H. Padua and M. Wolfe, "Dependence Graphs and Compiler Optimizations," Conference Record of Eighth Annual ACM Symposium on Principles of Programming Languages, Williamsburg, VA., January 1981,

14. P. Mehrotra, J. R. Van Rosendale, "The BLAZE Language: A Parallel Language for Scientific Programming," Report No. 85-29, ICASE, NASA Langley Research Center, Hampton, Va. (May 1985). (to appear in Journal of Parallel Computing).

15. D. Padua and M. Wolfe, "Advanced Compiler Optimizations for Supercomputers," Communication of ACM, Vol. 29, No.12, Dec. 1986, pp. 1184-1201.

16. G. Phister, A. Norton, "Hot Spot Contention and Combining in Multistage Interconnection Networks," Proceeding of the 1985 International Conference on Parallel Processing, IEEE 1985, 790-797.

17. C. Polychronopoulos, "On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems," Ph.D. Thesis, University of Illinois Center for Supercomputer Research and Development. CSRD TR.595, Aug. 1986.

18. Wang, K.-Y., Gannon, D., "Applying AI Techniques to Program Optimization for Parallel Computers," in **AI Machines and Supercomputer Systems** . Hwang, DeGroot, eds. McGraw Hill, NY, 1987.

19. M. Wolfe, "Optimizing Supercompilers for Supercomputers," Ph.D. Thesis, Dept. of Computer Science, University of Illinois, Urbana-Champaign, 1982.

END
DATE
FILMD
3-88
DTIC